Sensor Network Motes:

Portability & Performance

Leopold, Martin

Publication date:
2008

Document version
Publisher's PDF, also known as Version of record

Sensor Network Motes:

Portability & Performance

Ph.D. dissertation by Martin Leopold

Department of Computer Science
Faculty of Science
University of Copenhagen
January 2008

This document is typeset in LaTeX 2$_\varepsilon$.

It contains 104 pages, app. 47309 words and app. 307625 characters (excluding table of content, biblography, etc.). It is printed on 121 A4 sheets.

This document was compiled on Mon Dec 31 17:20:04 CET 2007 .

Abstract

This dissertation describes our efforts to improve sensor network performance evaluation and portability, within the context of the sensor network project Hogthrob. In Hogthrob, we faced the challenge of building an sensor network architecture for sow monitoring. This application has hard requirements on price and performance, and shows great potential for using sensor networks. Throughout the project we let the application requirements guide our design choices, leading us to push the technologies further to meet the specific goal of the application.

In this dissertation, we attack two key areas related to the design of this solution. We found the current state of the art within performance evaluation to be inadequate and that the moving to the next generation platforms is being held back by practical issues in porting existing software. We have taken a pragmatic, experimental approach to investigate these challenges and apart from developing the methodologies, we also present the results of our experiments.

In particular, we present a new vector based methodology for performance evaluation of sensor network devices (motes) and applications, based on application specific benchmarking.

In addition, we present our results from porting the highly popular sensor network operating system TinyOS to a new and emerging system on a chip based platform. Moving the sensor network field towards the use of system-on-a-chip devices has large potential in terms of price and performance.

We claim to have advanced the current state of the art within sensor networks within the two key areas: portability and performance.

Acknowledgments

I would like to thank professor David Culler for hosting me during my six months research visit to UC Berkeley. Arch Rock for providing me with facilities and equipment. I would like to thank the Hogthrob partners, in particular Cécile Cornou and Klaus S. Madsen. And finally I would like to thank my supervisor Philippe Bonnet, Eric Jul and Marcus Chang from DIKU.

My research grant is made possible through the Danish Technical Research Council[1]

[1]Statens Teknisk-Videnskabelige Forskningsråd (STVF)

Contents

CHAPTER 1

Introduction

This dissertation describes our efforts to improve sensor network performance evaluation and portability, within the context of the sensor network project Hogthrob. In Hogthrob, we faced the challenge of building an sensor network architecture for sow monitoring. This application has hard requirements on price and performance, and shows great potential for using sensor networks. Throughout the project we let the application requirements guide our design choices, leading us to push the technologies further to meet the specific goal of the application.

In this dissertation, we attack two key areas related to the design of this solution. We found the current state of the art within performance evaluation to be inadequate and that the moving to the next generation platforms is being held back by practical issues in porting existing software. We have taken a pragmatic, experimental approach to investigate these challenges and apart from developing the methodologies, we also present the results of our experiments.

In particular, we present a new vector based methodology for performance evaluation of sensor network devices (motes) and applications. The methodology uses a benchmarking approach to create an objective description of a mote, and further traces an application to extract an abstract workload descriptions from a running application. Combining these two are able to speculative estimate the performance of an application across motes.

In addition, we present our results from porting the highly popular sensor network operating system TinyOS to a new and emerging system on a chip based platform. Moving the sensor network field towards the use of system-on-a-chip devices has large potential in terms of price and performance.

We claim to have advanced the current state of the art within sensor networks within the two key areas: portability and performance. Before we further detail our motivation and define the problems covered in this dissertation, we introduce the Hogthrob project.

1.1 The Hogthrob Project

The work presented in this dissertation is part of Hogthrob research project. Named after the captain of the Muppet Shows "Pigs in Space", the Hogthrob project is a four year research project (started in February 2004). The goal is to build a sensor network infrastructure for sow monitoring. The project is a collaboration between three research institutions, and two industrial partners.

The key idea in the project is the use of sensor network technology within the area of Sow Monitoring. What can be observed and how? DIKU has focused on the sensor network infras-

tructure for this task. This dissertation is a continuation of my Masters Thesis and I will quote the introduction of the Hogthrob project:

1.1.1 Sow Monitoring

Current sow monitoring equipment is based on RF-id ear tags and readers located at feeding stations. This equipment has some advantages (its main goal is to control how much food sows are eating) and a number of drawbacks:

- When looking for a given pig, the farmer has to place a hand-held reader close to an animal—for large groups this can be time consuming. Legislation is underway that will require farmers to let sows roam freely in large pens. This will expose this problem further.

- Correctly establishing the onset of *estrus*[1] (heat period) is a major issue for pig production. The sows exhibit clear physical signs when the event occurs. Finding the exact moment can be done purely by observation or augmented by using a detection system.

  The available detection systems today rely on the fact that the sows are likely to approach a bore more often (if one is available) during the heat period. Placing a bore in an adjacent confinement and detecting the RF-id tags of the sows that approach it will provide a decent indication. However, sows are housed in groups with a strict hierarchy. A sow low in the hierarchy is unlikely to approach the bore. A purely RF-id based system will thus not detect the beginning of a heat period for all pigs.

Implementing a sensor network by placing a sensor node on each sow provides new insights and new solutions to the problems above.

1.1.2 Application Requirements

Monitoring sows in a large pen on a farm presents a concrete sensor network monitoring application with many interesting challenges. The application requirements are imposed by the farmers, not as a result of our imagination. The three major constraints are *price*, *life time*, and *form factor*:

1. The profit margin of sow production is low and the equipment for each sow must be very cheap in order to fit the budget (in the order of a few €).

2. The usability of the system on the farm will be drastically reduced if the nodes have to be manually inspected too often. A lifetime of as much as "a few years" would be advantageous, but in practice a lifetime of 6 months would be acceptable. The identification systems used today are not maintenance free as the sows tend to loose or eat the tags.

3. Ear-tags are a good trade-off for sow monitoring equipment. This form factor offers good guarantees in terms of robustness (pigs fight a lot and large equipment is likely to be damaged or lost), it doesn't hurt or annoy the animals and it is easy to install and remove (as opposed to injecting capsules in the body of a sow).

[1] Estrus is the period when a sow can be bred, and it lasts for a short time only. If a sow is not bred during its first estrus, it is considered unproductive from the commercial point of view since it will be another three weeks before estrus reoccurs. Meanwhile it needs to be fed and housed.

1.1.3 Project Challenges

Building a sensor network architecture that meets the requirements of the application is the objective of the project. The question is how? We choose to follow the following application driven approach:

- We choose a large design space that encompasses both hardware and software components and the interaction between them.
- We want to explore the design space and take design decisions based on how they contribute to meeting the application requirements.

Let us first discuss sensor network motes and then define the problems we undertake in this dissertation.

1.2 Sensor Network Motes

Sensor networks-based monitoring applications range from simple data gathering, to complex Internet-based information systems. Either way, the physical space is instrumented with sensors extended with storage, computation and communication capabilities, the so-called motes. Motes run the network embedded programs that mainly sleep, and occasionally acquire, communicate, store and process data.

The overall goal of the Hogthrob project is to design a sensor network architecture that meets the requirements of the application (Section 1.1.2). The question is how? The approaches we have seen application builders take, can roughly be divided into two: build a mote or buy a mote. Buying a mote relies on having a *generic* mote available that, while being general enough for several applications, also performs well adequately. Alternatively a *custom* mote could be build specifically for an application, ensuring that the performance is sufficient.

Regardless of the approach to acquiring a mote the, the key question is will it perform well enough? We refer to the process of answering these questions as *exploring the design space*.

1.2.1 Exploring the Design Space

By exploring the design space we denote the process of evaluation design options in relation to a set of criteria. Depending on the application at hand the depth of this process may vary, but it is our claim that all sensor network applications will require some form of exploration. The Hogthrob project was started with an exploratory phase, in part consisting of a pilot experiment, the purpose of the process was to understand the application at hand at a more fundamental level and to learn the possibilities and limitations of potential solutions. We believe this process is a common trait for all sensor network applications, and it is our claim that this process is essential to producing an efficient solution.

The design options available in the sensor network domain are very open. For many projects designing a new mote is not out of reach, while most opt for an existing design. An existing design has many appealing qualities, in particular saving the time consuming task of building and designing a mote. Using an existing mote design will enable us to implement an application solution much more rapidly than having to design one from the ground. Essentially generic motes may bring the advantages of a PC to the sensor network domain—cheap, flexible and easy to use. Within the research community a large number of projects has been focused

on building generic motes (one-size-fits-all), and a large number of operating systems, motes, network infrastructures are available to us within this domain.

The alternative to a using an existing mote is to design a new mote for a given application. By building a mote specifically for a set of requirements allows a much higher integration with the features of the application, and possibly taking advantage of certain performance potentials. One of the most promising visions to facilitate a new level of performance is the use of hardware accelerators. A hardware accelerator is a certain piece of hardware that enhances performance by alleviating the system. To build such an accelerator requires intricate knowledge of the performance bottlenecks.

While promising gains in terms of price and performance, the use of application specific motes is very little. Both in the research community and commercially the norm at this time is to rely on generic motes.

Generic Motes

In order to increase reliability and reduce complexity, research prototypes [34, 75] as well as commercial systems[2] now implement a tiered approach where motes run simple, standard data acquisition programs while complex services are implemented on gateways. These data acquisition programs are either a black box (Arch Rock), or the straightforward composition of building blocks such as sample, compress, store, route (Tenet). This approach increases reliability because the generic programs are carefully engineered, and reused across deployments. This approach reduces complexity because a system integrator does not need to write embedded programs to deploy a sensor network application.

Such programs need to be portable to accommodate different types of motes. First, a program might need to be ported to successive generations of motes. Indeed, hardware designers continuously strive to develop new motes that are cheaper, and more power efficient. Second, a program might need to be ported simultaneously to different types of motes, as system integrators need various form factors or performance characteristics.

1.3 Problem

In the context of the Hogthrob project we focus on application specific motes, due to the extreme performance requirements. The key question for Hogthrob is: does a given mote design meet the requirements of the application. For us this means that we must perform a set of exploratory experiments that gives us the insights required to design the final mote. The result of such a process could be that a generic mote is sufficient or that wee need to build a new mote with certain properties. Regardless, the question remains how to evaluate the performance of the mote designs in relation to the requirements.

In our view this is a general problem in sensor network mote design—how do we compare performance across mote designs? The most prominent sensor network deployments rely on trial and error, and the benchmarking techniques that are available do not provide the insights into how a given application (or workload) will perform.

Whether using an existing mote or developing a new mote, having a portable software base will greatly reduce the complexity of implementing an application. As mentioned above, currently the most common approach relies on the paradigm of implementing a sensor network using generic motes. Each mote runs the same program and is configured on the fly using some

[2] See http://www.archrock.com

infrastructure. Moving such an infrastructure to a new platform that better suits the application requires that the software base i portable.

In this dissertation we focus on performance evaluation and portability.

1.3.1 Performance Evaluation

For us performance evaluation means, answering the question: will this mote perform adequate within the parameters required by our application? We observe that the current evaluation methods are inadequate in a number of key parameters. To uncover the answer to this question we need a methodology to evaluate the relevant parameters.

Benchmarking A common approach to understanding the performance of an application on new hardware platform is to look at a benchmark. A benchmark is run on a number of possible candidates, and the winner is the machine that best fulfills the benchmark. The problem with this approach is that relating the workload to the benchmark to the workload of an actual application is difficult.

Earlier attempts at creating sensor benchmarks have taken a traditional approach[41]. Attempting to create *stress marks* that give insights to how two platforms relate to each other.

The problem with this approach is, as with benchmarks in general, that relating the performance of the benchmark to the performance of an actual workload is difficult if not impossible. In the case of sensor networks the problem is further apparent: the energy consumption is entirely dependent on the particular workload that is imposed by the application.

Prototyping approach Building a new mote often starts with a prototype. Such a prototype can serve many purposes, a starting point, a learning platform, etc. However, as a prototype the major purpose is to develop a final mote.

No current method allows a designer to reason systematically about the performance of the final mote using the prototype mote.

Measurement methods do not span platforms At the current stage in sensor networks all options are open: hardware, operating system, applications, etc. A general method must span all of the available design options.

Subsystems are evaluated in isolation The process involved in determining the performance of a sensor network mote cannot be viewed in isolation. Consider for example the impact of a radio, such a component has a data sheet, which should determine the performance. Running a radio requires a power supply, it requires a CPU to perform certain computations, etc. It is essential that these overheads are part of the performance equation.

Simulation is insufficient to model the environment Performance evaluation is often based on simulations. When the environment is not well understood, or the impact of the environment on performance simulation, is insufficient.

The time-frame is too short In our case we wish to expand the time frame to reason about the entire lifetime of our deployment, lasting days, months or years. Current methods only look at short time intervals we need to expand the time frame to cover the entire period.

1.3.2 Portability

As we have discussed the sensor network design process may greatly benefit from portability. While the issue of portability has been the focus of a large body of research in the sensor network community, the number of platforms still remains relatively low.

TinyOS As one of the popular sensor network operating systems the platform support of TinyOS impacts the field as a whole. TinyOS is designed for high portability and the recent version 2 of TinyOS introduces a stringent layered approach. Still, the supported platforms are largely the same as previous versions of TinyOS and other sensor network operating systems.

Prototype Designing a new platform using an existing platform will greatly benefit from a portable software environment. In particular the vision of adopting *hardware accelerators* could be archived by the use of a prototyping approach.

Low platform adoption While interesting platforms are emerging promising superior performance, only a handful of platforms are widely adopted. We claim that the practical issues are holding the adoption of new and interesting platforms back. In particular the emergence of system-on-a-chip based devices, has not caught on despite the fact that large performance gains are within reach.

1.3.3 Thesis Statement

In this dissertation we focus on two key issues that are currently holding the continued evolution of sensor networks back

Performance evaluation Current sensor mote evaluation methods are insufficient. It is essential to develop new methods that allows us to compare the performance of applications across mote designs.

Portability Current sensor network support systems claim to be highly portable and well suited for a broad set of hardware platforms. In particular the recent TinyOS 2 has redesigned the underlying architecture to better facilitate portability. To a large extent, this claim is untested.
We wish to investigate this claim, by picking a fixed point in the design space (the 8051) and provide a proof of concept implementation.

1.4 Approach

The area of sensor networks has for some time been the subject of a large research effort. While envisioning a bright future for sensor networks, many project fall short in the tangible solutions. Therefore our approach is pragmatic. We wish to transform the visions into concrete implementations. We will implement proof of concept solutions and conduct the necessary field studies to support our claims.

The first task at hand is a practical approach to the question: how do we explore the design space? One of the most promising paths in sensor network mote design is the use of system-on-a-chip design. This technology has been on the verge of a breakthrough in sensor networks for

some time, but the barriers have never been broken. To get access to the this technology we pick a fixed point in design space: the Texas Instruments CC2430. We want to explore the challenges and the potential of this system on a chip in the Hogthrob application. This platform will serve as a *prototype* to the final platform.

The second task is the evaluation. As described previously we find the existing evaluation frameworks inadequate. We design and implement a evaluation method that allows us to find general characteristics of a mote and of an application.

1.4.1 Work Done

Within the Hogthrob project we concentrated on the process of providing a feasible sensor network design for sow monitoring. One key element in this was understanding the application. We took part in the team effort that resulted in the first pilot experiment collecting data from sows during their ovulation period. The experiment is described here [19, 40, 66].

- The field studies have been carried out as a collaboration between DIKU and LIFE. The implementations were provided by DIKU and I assisted in practical aspects, the overall design process and the following data scrubbing phase.
- In collaboration with two master students we designed a 30 day experiment to be setup in a stable. This setup contained two redundant servers collecting video and acceleration data transmitted from the sows using Bluetooth.
- The setup failed in a number of ways. Combining the collected data and video data from the two redundant servers turned out to be non trivial and time consuming.

In relation to the sensor mote design we investigated two paths to gain access to a system-on-a-chip: build one and buy one. By buying a completed SoC, we benefit from a chip that, although not tailored to our approach, provides the characteristics intrinsic to SoC design. Producing a chip is beyond the capabilities of the Hogthrob project, but to be able to explore the possibilities available from a custom built SoC we created the HogthrobV0 prototype platform. While this platform does not provide the energy performance of a SoC it can be configured to be functionally equivalent to one.

- The next step in the use of system-on-a-chip design is the use of hardware accelerators. To explore this path the HogthrobV0 platform, we have used to mimic the functionality of cc2430 - by using the freely available Oregano 8051 core and the on-board radio. This setup allows us to carefully measure the impact of adding additional hardware to the architecture.
- The HogthrobV0 platform will enable *in-situ* experiments of hardware accelerators. Instead of relying on feeding a hardware simulation with a model of the environment, this platforms allows hardware simulation with real input.

Using the HogthrobV0 platform as a prototype introduces two problems: i) it does not have the performance of the platform we are trying to simulate and ii) the software needs to be portable to be translated from the prototype to the final mote. TinyOS has for some years been promising to deliver highly portable, cross platform, sensor network operating system. We test this hypothesis with a proof of concept implementation for 8051 based platforms.

- I spent a 6 month research visit at the University of California Berkeley, working with some of the driving forces behind TinyOS, this greatly sped up the process of implementing TinyOS for 8051 and in learning the motivation behind TinyOS 2.

To be able to speculatively forecast the performance of a final mote, before it is available we adapted the vector based methodology, initially proposed by Setlzer et al.[92], to study mote performance in general and TinyOS-based motes in particular. We implemented this method on two commercially available sensor motes: Sensinode Nano and Sensinode Micro, and we experimentally verified the viability of the method.

- First, we test the hypothesis underlying our approach.
- Second, we compare the performance of the Micro and CC2430 motes using their hardware vectors.
- Finally, we predict the performance of generic data acquisition program from Micro to the CC2430.

To gain a broader perspective of the applications that may benefit from the use of sensor networks I have taken part in a number of events.

- I took part in TinyOS Technology Exchange 2005, 2006, conference participant IPSN'07, SenSys 2005, 2006, 2007. I was paper reviewer for SenSys 2005 and 2006.

  I met with industrial partners through Danish Industries[3], and the agricultural fare "Down to Earth" (Jordforbindelse)

  As well as student project supervisor, internal censor, guest lecturer, etc.

1.5 Contributions

We claim to have advanced the field of sensor networks in the following areas:

A new methodology for sensor mote performance benchmarking

We have proposed a methodology for comparing and benchmarking sensor network motes. This methodology provides will aid sensor network designer answer key performance questions, when designing or selecting sensor network motes.

An experimental verification of our performance methodology

We have implemented our method in TinyOS 2 for two commercial sensor network motes: Sensinode Nano and Sensinode Micro. We present the results of our experiments.

A quantitative performance comparison of two sensor network motes

We present a quantitative comparison of Sensinode Nano and Sensinode Micro using our vector based benchmark.

[3] Confederation of Danish Industries (Danish: Dansk Industri)

A highly compatible port of TinyOS 2 for the 8051

We present an implementation of the operating system TinyOS for the 8051 platform. The 8051 architecture differ substantially from the architectures used to develop TinyOS. We have ported the sensor network operating system TinyOS 2 to the CC2430 and 8051 platforms.

We propose a method that allows a single source tree to be compiled using different compilers with different C-dialect. We believe this is the first 8051 operating system to support multiple compilers using a single source base.

A comparison of the CC2430 and the Sensinode Nano

We further detail the comparison of the CPU benchmark used as part of the vector based methodology. This comparison complements the results based on the vector based approach, detailing the impact caused by the architectural differences.

Lessons for portable sensor network operating systems

We present our lessons from porting TinyOS to the 8051 platform. These solutions are not limited to TinyOS or the 8051 and we consider the lessons to be generally applicable to other projects facing similar challenges.

The lessons consist in part of the work presented here and of our guide to building TinyOS 2 platforms published separately as a technical report[58].

1.5.1 Technical Contributions

A Platform Enabling Hardware/Software co-design

- HogthrobV0 prototype platform. We have implemented, tested and documented the Hogthrob prototype platform - an compact, flexible FPGA based prototyping platform. This platform allows a much wider design space for future researchers attempting to verify assumptions by field experiments. A manual for this platform is published separately as a technical report[59].
- Oregano for HogthrobV0. We have adapted a freely available 8051 core for the HogthrobV0.

1.6 This Dissertation

In this chapter we have detailed some of the motivation for this dissertation and the Hogthrob project. We have defined the two problems we address. In the following chapters we detail the work introduced here. Each chapter has been written to be self-contained in such a form that they can be read independent of the other chapters. Chapter 3 forms the basis of a publication at the EWSN 2008 conference, and Chapter 4 has been written such that it could form the basis of a publication at a later date.

We will begin by building some of the background for the Hogthrob project in Chapter 2. Here we will cover some examples of sensor network applications, review some of current

sensor mote hardware platforms and look at the power estimation techniques available to us. We will also look at the HogthrobV0 prototype platform, that we designed in the project.

Chapter 3 introduce performance evaluation in sensor networks and present our vector based methodology. Chapter 4 discuss portable operating systems and present our methodology. We will conclude our findings and summarize the results in Chapter 5.

CHAPTER 2

Background

The following chapter builds some of the general background for the Hogthrob project. This chapter builds the knowledge required for the Hogthrob project, and serves to illustrate the context of the project, but is not strictly focused on the problems we describe in this dissertation. The context described here was important to us and the project, but some sections are less relevant to the points of this dissertation.

We direct readers that are not particularly interested in the review of mote design and power estimation techniques to skip to Section 2.5 covering the Hogthrob prototype platform.

In Section 2.1 we begin by describing two classic examples of sensor network deployments. The two examples are used to monitor animals in two quite different scenarios. We compare this to the monitoring application of the Hogthrob pilot experiment. The intention is to build a context for the remainder of the discussions, in particular the kind of challenges that we are ultimately facing. The focus of the section is how the requirements of the application are met with the design of the architecture.

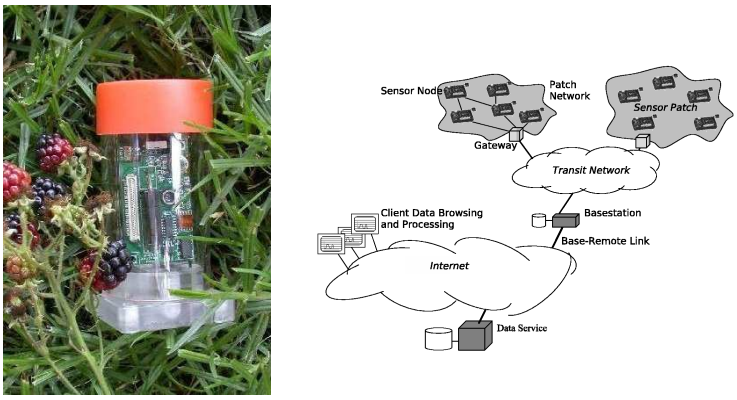
In Section 2.2 we survey past and current generations of sensor network motes. The review is based on the motes available on at the beginning of the project and is not up to date with the most recent developments. We use this section to build the case for the need for a specialized design, that we choose as the goal of the project.

In Section 2.3 we describe some of the approaches chosen to program sensor networks in general. We describe some of the related work to TinyOS and outline the design principles behind TinyOS that differ substantially from other systems. This discussion is important to the context TinyOS to sensor networks.

In Section 2.4 we review some of the related work regarding power estimation. The related work is not directly related, but it serves to build the intuition required for our power estimation technique in Chapter 3.

In Section 2.5 we present the HogthrobV0 prototype platform. This platform lays a foundation for further exploration of the hardware software boundary, but this work is far from complete. We include it here as a technical contribution. The potential of the platform include combining this platform with our vector based methodology (Chapter 3) and experimenting with the hardware software boundary by implementing hardware extensions to an 8051 CPU running on the embedded FPGA.

The motivation for the Hogthrob project has been explored previously, and major parts of this chapter has been published previously [ML04].



(a) Housing (b) Network Architecture

Figure 2.1 *Great Duck Island. Mica mote in acrylic enclosure and schematic network architecture[84].*

2.1 Sensor Network Monitoring Applications

Recent years has seen a number of scientific and commercial applications of sensor network monitoring. For scientific monitoring, sensor networks have provided a number of advantages over current methods in several ways. Some project have greatly benefited from on board computing, spacial allocation, in network processing, etc.

We will look at a couple of scientific examples and focus on how the requirements of the application are met.

2.1.1 Great Duck Island

During 2002 researchers from University of California Berkeley (UCB) and The College of the Atlantic deployed a sensor network with 32 nodes on a desolate island off the coast of Maine. The goal was to monitor the habitat of a small sea bird, the Leach's Storm Petrel. The target lifetime was to monitor the birds during their 7 months breeding period. The sensor network monitors how the birds use their burrows and it monitors the micro climate in them. The Leach's Storm Petrel and other seabirds are sensitive to disturbance—sensor nodes provide a low invasive alternative to frequent visits[72].

Sensor Network

The deployed sensor nodes were slightly modified Mica motes (see section 2.2.2) equipped with the Mica weather sensor board[1]. The weather board features temperature, photo-resistor, barometric pressure, humidity, and passive infrared (thermophile) sensors.

To withstand the harsh, outdoor environment, sensor nodes are covered with a thin parylene sealant which protects exposed electrical contacts from water. The on-board sensors remained exposed to preserve their sensitivity. The nodes were placed in a ventilated acrylic enclosure (see Figure 2.1(a)).

[1]Manufactured by Crossbow http://www.xbow.com

The nodes were spread out over a 15 acre area and results were forwarded to a central database. The wide spread of the sensor nodes demands a sophisticated network infrastructure. The authors choose a two tiered architecture by grouping sensor nodes close together in a *sensor patch* with a gateway that is part of a *transit network* that transmits the data to a remote data storage unit (see Figure 2.1(b)).

As a simple health sign the nodes regularly included their battery voltage with their sensor reading. This measure assisted researchers in analysis of remote node failures and provide insights in deviating sensor reading.

The authors consider this application is representative of a class of sensor network applications described as *habitat and environmental monitoring*, with the following characteristics:

- Immobile nodes that are left unattended for long periods of time.
- On-line data gathering, measurements are forwarded through network infrastructure

While the experiments were planned for as long as 7 months many of nodes failed much earlier than this. Interestingly only a few died because of depleted batteries, the majority failed to withstand the wear and tear from the outdoors. Based on this fact node failures are shown to be predictable based on their, faulty, sensor readings. An other surprise was the networking performance the nodes send infrequently and at a low rate, suggesting few or no collisions. However, in the deployment it turns out that by different types of misfortune the nodes start dropping a large number of packets for example at certain period the transmission schedule is aligned and packets collide[98].

Lessons Learned

The Berkeley team have learned lessons from their experiments in a number of domains ranging from packaging to network protocols. From our point of view the most interesting lessons are:

- The differences in conducting lab and field experiments
- Their approach consist in using pre-designed hardware and package, so that it can fit in a burrow and survive outdoor conditions. They suggest that a more effective approach would be to account for environmental conditions and specific sensors when designing hardware and software.

2.1.2 Zebranet

In January of 2004 the Zebranet project monitored herds of Zebras roaming freely in the plains of Kenya[49]. The goal of the project is to conduct a live experiment attaching collars with sensor nodes on herds of Zebras and log their position using GPS during one year.

Some 35,000 Zebras roam freely in the 40,000 km^2 Laikipia plateau of central Kenya in larger or smaller groups depending on their species. The speed and direction of movement of the individual animals in a group is closely correlated, thus tracking an entire herd can be accomplished by collaring only a single or a few animals in a group, vastly reducing the number of collars required.

Sensor Network

Traditional tracking is based on collaring animals with VHF transmitter and locating the animals by driving through or flying over the expected locations listening for “pings” from the transmitters. The freely roaming Zebras give rise to a radically different scenario than the Great Duck Island scenario:

- The nodes are mobile
- The base station is mobile (moving along with the researchers camp)
- The nodes are not in contact with a base station or network at all times

It is unattractive to deploy fixed infrastructure through out the park mainly because of the risk of vandalism and the large area. To solve these problems the authors observe that the herds of Zebras tend to meet regularly at water-holes scattered throughout the park. Using this observation, they choose a peer-to-peer *data dissemination* strategy (similar to Manatee[5]): measurements are replicated from node to node when they are within radio range an to the base-station when it is in range. By using the last time of contact with the base station as a data replacement heuristic, the measurements will statistically make their way towards the base station.

Prototype Nodes

The authors present multiple generations of prototype nodes, from the first proof of concept (version 0.1[49]) to a small integrated platforms powered by solar cells (versions 1,2, and 3[105]). The prototype platform experimented with a dual radio system for long / short range communication, but this was not used in the field, as the authors believe that it was unlikely the dual range principle was of much use. In January 2004, a batch of the version 3 nodes were deployed in Kenya, a summary of the features is given in Table 2.1 on page 19.

The platform uses a GPS receiver to obtain the location at regular intervals and logs this in the on-board flash. They choose a long range, low data-rate radio (MaxStream 9xStream[2]). The GPS unit and the radio are high power devices (compared with the devices we will look at in Section 2.2) and to sustain its power budget the platform recharges a battery using solar cells embedded in the collar.

It is also noteworthy that, although the authors point to many inefficiencies and power optimizations in the platform, it turned out to be *good enough*—12 zebras were collared with and the platform functioned autonomous on the plains of Kenya. A few preliminary results have been published, but detailed results from the deployment is not available at this time[105].

Lessons Learned

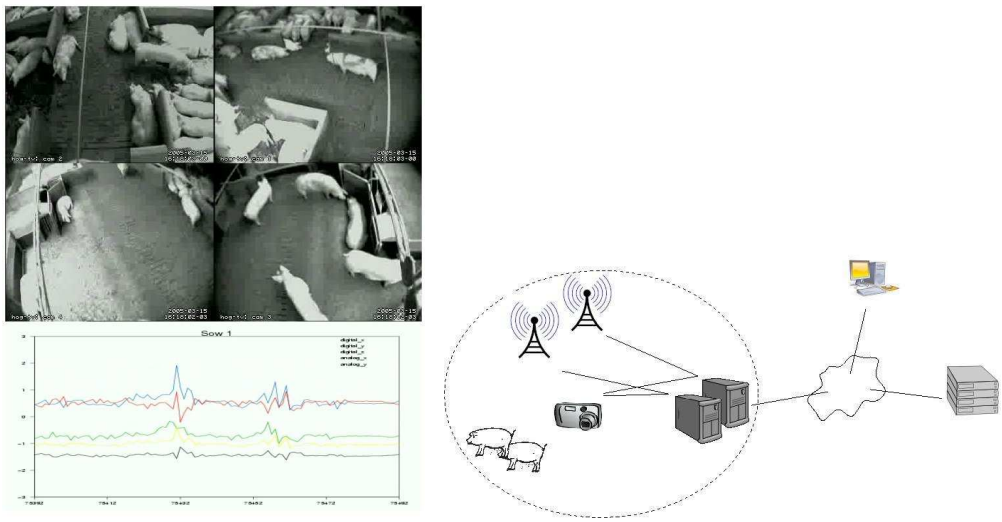
The authors gain insights into how to design a sensor network platform and how to conduct experiments in the field. The lessons we take from the Zebranet deployment are:

- The authors developed *specific* nodes driven by the requirements of the application. In this case the requirements included long range radios, weight, size and GPS-logging.
- The new platform was fixed first and then software was developed. There was no evaluation of how the hardware could best support the software.

2.1.3 Hogthrob Pilot Experiment

The Hogthrob project, as mentioned earlier, involves building a sensor network for monitoring sows, and detecting heat in particular. The Hogthrob project begun with an exploratory phase, consisting in part of a pilot experiment. The purpose of this experiment was to verify earlier findings regarding sow behavior using group housed sows, as opposed to individually housed

[2]http://www.maxstream.net



(a) Data (b) Setup

Figure 2.2 *Pilot experiment at Askelygaard. a) the 4 cameras and a plot of the data collected from one sow. Notice that one of the five sows we are monitoring is in upper right frame with markings on the back, b) a depiction of the setup two PC are connected to a Bluetooth receiver and all 4 cameras are connected to both PC. The PC offload their data to remote servers via the Internet and allows remote monitoring.*

sows, as well as gaining familiarity with setting up experiments in a stable. The data would further aid in building a model of the sow behavior.

The experiment consisted in monitoring five sows before, during and after their heat period including the time of ovulation. During this time the sows are fitted with a sensor collecting acceleration data. In addition data, ground truth is collected using cameras and manual observation of physical trait indicating heat. An visualization of the cameras and acceleration is depicted in Figure 2.2(a).

The heat period of the sows lasts only for one or two days and occurs approximately once a month. The five sows are taken out of their regular production cycle and monitored for a 30 day period. This period should collect data in a non-heat and a heat period and take into account the irregularity of the ovulation.

The experiment is characterized by:

- Deployment for a short period.
- Collecting data and learning the nature of the application are equally important goals.
- The environment is different than, what would traditionally be considered a sensor network: while the sender is battery powered, the infrastructure is connected to the power grid.

Sensor Network

The sensor network is built using a Bluetooth enable sensor mote sampling acceleration to a local buffer offloading it to a server once every hour (See Figure 2.2(b)). The mote contains two accelerometers a digital and an analog that are sampled at 4 Hz. The data is offloaded to one

of two servers that receives and time stamps the data. The data is stored locally and forwarded via the Internet to a central storage.

The store and forward strategy is chosen to accommodate the long connect time, but high data transfer rate of Bluetooth. In this way, the energy pr. bit becomes relatively low. The video is recorded on two servers for redundancy if one server fails. We choose a get all strategy: collecting all data from the motes and doing all processing offline.

Lessons Learned

The experiment was a valuable lesson in building the final sensor network infrastructure. The collected data was not perfect, but provided enough value to get started on a model. In all the pilot was success and create a foundation for further experiments. In total 240 MiB of sensor data was collected, along with 30 days of video.

- Communication failures were much more frequent than expected. Quite possibly because the sows might lie down on a mote in such a way that it could not of load the data. This caused extended periods of missing data.
- One of the accelerometers was misprogrammed and returned faulty values.
- The video servers did in fact fail at random, however, joining the video turned out to be time consuming.
- The time synchronization for each server was misconfigured. Meaning that no continuous time exist for all data, because the motes are equally likely to check in on one or the other server.

Validating the assumptions prior to the experiment was not carried out - checking that the accelerometers return correct values and that the connections were sufficient (even with sows lying on the motes). Prior to the experiment no planning was made for the aftermath - the data scrubbing after the experiment was much larger than anticipated.

Many of these faults could have been cured by more careful planning and higher focus on getting meaningful data from day one.

2.1.4 Discussion

We looked at three examples of sensor network deployments that were deployed early in the project phase. In all tree cases the problem was not well understood and the experiments lead to new insights into the problems. The three examples used a low level of in network processing and off loaded as much data as the network could handle. We believe these three can be considered as prime examples of pilot projects, and serve as examples to the development process for other similar projects: starting out with a prototype design and a get-all data collection strategy.

Looking back, we can distinguish two types of requirements that lead to the design of these sensor networks:

Functionality sensing capabilities, modularity, data collection / dissemination

Performance lifetime, price, energy budget, form factor, environmental resistance (rain, fumes, etc.)

Based on these requirements choices were made to design a sensor network—as far as the design decisions are concerned, we make the following observations:

- The Great Duck Island designers chose to use pre-designed sensor nodes while the Zebranet designers chose to define their own sensor nodes.
- In both cases, the application requirements were met through a trial-and-error approach that consist of a) a pre-deployment analysis (either back of the envelope calculations or micro-experiments in the lab), b) an on-line monitoring (battery level indication) and c) a *post mortem* analysis (that rely on data logged during the experiment).

More generally the examples show the challenges that are faced within the type of applications that we denote as sensor networks. Constrained in size, energy and price and requiring some form of network communication possibly employing in-network aggregation.

In the context of Hogthrob, a first question is whether to use a generic, pre-designed sensor node or to design our own. As far as meeting the application requirements (described in Chapter 1), we aim at following a systematic approach for which this thesis is a foundation.

In the next section we will try to place the existing platforms in relation to the design parameters above.

2.2 Sensor Network Platforms

In recent years there has been growing research in the field of building sensor network platforms, each of these platforms are a point in the design space. Most of the platforms are used to investigate a multitude of research topics ranging from network issues, remote reprogramming, sensing capabilities to software design or scalability. Only a few platforms have been evaluated in the context of field experiments.

The major drawbacks to designing and building sensor nodes, disregarding the cost is: (a) the design process itself is a long and time-consuming process and (b) scaling a network to hundreds or thousands of nodes is difficult. To overcome this initial hurdle and to study large scale sensor networks simulation is often employed. We will come back to the topic of simulation in Section 2.4 when discussing power estimation, but using simulation as a sensor network platform will not give us insights to the possibilities in hardware design that exists today.

The question we posed was: is there a platform available today that we can use in the Hogthrob project? In this section we look at the available sensor nodes.

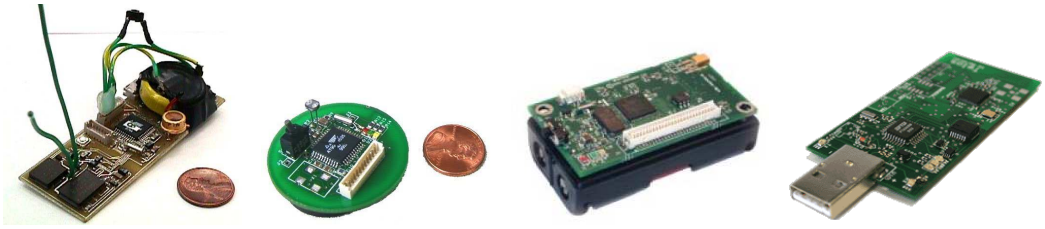
The nodes we describe in Section 2.2.2 have been built using commercially available or common off the shelf components (COTS), the next natural move for sensor network platforms is to embed all components on a chip (system on a chip). We look into two projects exploring this practice in Section 2.2.3.

2.2.1 Sensor Network Motes

The sensor networks we see today are characterized by instrumenting the physical world using motes. The motes in turn run the embedded programs that collect, store and transmit the measurements collected by the motes. Some networks are composed of homogeneous motes, while some adopt a tiered approach with heterogeneous motes at different levels.

2.2.2 Generic Sensor Nodes

By generic nodes we mean nodes that are built to fit a general picture of a sensor network node and not specialized to a certain purpose. We look into a broad range of the generic sensor nodes available today, and compare them in Table 2.1



(a) RF Mote (1998) (b) weC (1999) (c) Mica2 (2002) (d) Telos (2004)

Figure 2.3 *Four generations of UC-Berkeley Motes*[4]

UC Berkeley Motes

The vast majority of research in sensor networks has been centered around the generations of sensor nodes developed at UC Berkeley: Rene, Mica, Mica2 [1, 2] shown in Figure 2.3—denoted as "motes".

Among the first motes to be developed at UC Berkeley were the "RF Mote" and the "weC" motes [46] featuring RF Monolithics TR1000 radio and the Atmel AT90LS8535 at 150 kHz and 4 MHz respectively. While the RF Mote has low power consumption, it is unable to operate the radio anywhere near its maximum capability. The AT90LS8535 is a *Harvard architecture*[3] without the ability to write in the program memory and therefore the motes contain a co-processor to handle reprogramming.

Building on the experiences of these nodes, the Rene and Rene2 were constructed in a modular design as a "sandwich" board, allowing easy and compact connection of additional boards (sensor board, etc.). The Rene2 featured the ATMega163 MCU at 4 MHz increasing the memory from 0.5 KiB to 1 KiB and the program flash from 8 KiB to 16 KiB.

The successor to these nodes was the Mica[44] and Mica2[64] motes continuing the modular design but upgrading to a more powerful MCU: the Atmel ATMega 103 at 4 MHz and ATMega 128l at 7.37 MHz respectively. Among other things the ATMega128l eliminates the coprocessor for writing to program memory. Based on the experiences in the first Great Duck Island deployment, the Mica2 and derivatives (Mica2Dot, MicaZ) are designed without a battery voltage up conversion (step-up or boost converter) and operate on unregulated battery voltage. As the battery is depleted, the voltage will drop affecting components such as the radio, sensors, etc. this influence has not been explored.

The Rene, Mica and Mica2 motes were (and are) commercialized by the spin-off company Crossbow[5] and are used by most sensor network research groups today. A number of variants have been manufactured such as the Dot and MicaDot, primarily with smaller footprint.

Telos

The Telos node from the latest UC Berkeley spin-off MoteIV[6] combines the Texas Instruments TI MSP430 with the Chipcon CC2420, 802.15.4 radio. The node does not feature the modular design of the Mica nodes, but has an on-board USB port for easy programming, making them

[3]The term *Harvard architecture* denotes the separation of program and data memory as opposed to *stored program* or *von Neumann* architectures in which program and data resides in the same memory space [42]

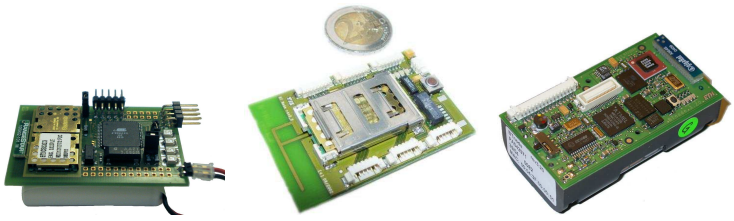
[4]http://www.tinyos.net/media.html

[5]http://www.xbow.com

[6]http://www.moteiv.com

	MCU	Clock (MHz)	FLASH (KiB[7])	RAM (KiB[7])	Wakeup (μs)	Storage (KiB[7])	Radio
RF Mote[46]	AT9080515	0.15	8	0.5		32	TR1000
weC[83]	AT90LS8535	4	8	0.5	1000	32	TR1000
Rene[83]	ATMega163	4	16	1	1000	32	TR1000
Mica[83]	ATMega103	4	128	4	180	512	TR1000
Mica2[83]	ATMega128l	7	128	4	180	512	CC1000
MicaZ[83]	ATMega128l	7	128	4		512	CC2420
BTNode2	ATMega128l	7.35	128	64		0	ROK101007
BTNode3	ATMega128l	7.35	128	184		0	ZV4002, CC1000
iMote	ARM7	12	512	64			Zeevo Bluetooth
Eyes	MSP430	5[8]	60	2		244	TR1000
Telos	MSP430	8	60	2	6	512	CC2420
ZebraNet	MSP430	8	60	2		3.8	9xStream
MC13192	MC9S08GT60	40[8]	60	4		0	MC13192

Table 2.1 *Sensor node summary. FLASH is used for program memory, Clock is the MCU clock, Power is the power consumption of the MCU. Storage is extra nonvolatile storage.*



(a) BTNode1 (b) BTNode2 (c) BTNode3

Figure 2.4 *BTNodes from ETH Zürich*[9]

ideal for educational purposes and less sensible to wear and tear when connecting and disconnecting plugs to external components.

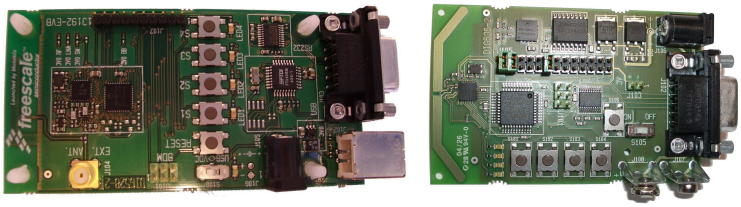
MicaZ

The latest Mica variant from Crossbow. As it predecessors it is based on the AtMega128l and has an external serial flash and as the Telos node it features the CC2420, 802.15.4 radio. The form factor is the same as the Mica nodes and it remains compatible with the Mica sensor boards.

BTNode

The BTNode generations of nodes have been developed at the ETH Zürich in the Smart-ITs project[10] (the three generations are shown in Figure 2.4). The Smart-ITs prototype[11] and BTNode2[8] were functionally equivalent, however the prototype was merely a proof of concept. They both feature an Atmel ATMega128l and an Ericsson ROK 101 007 Bluetooth module.

[7] KiB, MiB, and GiB is defined as 2^{10}, 2^{20}, 2^{30} bytes respectively[18].
[8] Variable up to
[9] http://www.btnode.ethz.ch
[10] http://www.smart-its.org



(a) MC13192-EVB (b) MC13192-SARD

Figure 2.5 *Freescale evaluation boards for the ZigBee-ready platform*

Additionally a 60 KiB external RAM block and a battery charge indicator in the form of a simple voltage divider is provided on-board.

Recently the BTNode3[7] was released, this node is developed at ETH, but manufactured and sold commercially by Art of Technology, Zürich[11]. It features the Atmel ATMega128l, 244 KiB external RAM and dual radios: Chipcon CC1000 and Zeevo ZV4002. In contrary to the BTNode2 design the BTNode3 has been designed in a sandwich fashion in order to allow easy connection of additional boards.

Eyes

The Eyes project[12] has yet to published details on their prototype nodes, however a short overview has been publish with the T-Mac radio medium access protocol[101]. The prototype features the 16 bit Texas Instruments MSP430F14 with 2 KiB RAM and 60 KiB flash, variable clock up to 5 MHz. Additionally it has an RFM TR1000 radio, and 2 Mbit EEPROM. The potential of the variable clock is not explored.

Intel Mote

The Intel IMote[52] is based on an Zeevo Bluetooth module with integrated ARM7 core (part number not available) with very few other components. The nodes are designed in a stackable fashion for easy connection to sensor boards. The details are few, but link reliability is argued as one of the advantages over more simple radios. An example deployment is described monitoring vibration in an factory scenario. The factory is a radio-hostile environment, with many obstacles and machinery generating noise. Even in this environment the Zeevo Bluetooth radio shows good connectivity and range.

Freescale Evaluation Boards

Recently DIKU acquired two different Freescale[13] evaluation boards featuring the Motorola 802.15.4 ZigBee-ready platform: the MC13192 Evaluation Board (EVB)[28] and the MC13192 Sensor Applications Reference Design (SARD)[29] (shown in Figure 2.5). Both feature the Freescale 802.15.4 MC13192 radio and the MC9S08GT60 microprocessor (part of the HCS08 family). The MC9S08GT60 is an 8 bit MCU with 16-bit addressing space and a variable clock speed up to 40 MHz, featuring 4 KiB RAM, 60 KiB FLASH, 8 ADC channels

[11] http://www.art-of-technology.ch
[12] http://eyes.eu.org
[13] A Motorola company http://www.freescale.com

The MC13192-EVB has a few push-buttons, LEDs, pin headers for external sensors and a USB or RS-232 programming port. The MC13192-SARD features the Freescale MMA6261Q, MMA1260D 1.5 g accelerometers.

Conclusion

When designing a sensor network platform using commercially available components the number of options is tremendous: MCU and radio manufacturers are plentiful. In this context it is surprising to see such a small diversity in choices. Even among the nodes designed by different research groups the choices are quite similar—no single node distinguishes itself from the others as remarkable.

Each of the nodes were the product of a certain design point, locking a design based on the available options. In each case when a hardware design was fixed the software was limited by the choices made in the beginning.

In general this approach allows flexible development, with add on sensors, easy component replacement, etc. The primary drawbacks to this approach is the constraints in terms of energy consumption and the form factor of the assembled printed circuitry boards (PCB).

2.2.3 System on a Chip

The sensor network deployments described in Section 2.1, and the available platforms described in the previous section are based on COTS nodes. Such nodes are easy to build or can be purchased commercially, but has a number of drawbacks. A solution to address these problems is to consider a sensor node on a chip i.e. assembling all components of a senor node on a single chip.

In the following we will look into a few projects, that while being very interesting, are still in their early stages and only being tested in simulation or lab experiments.

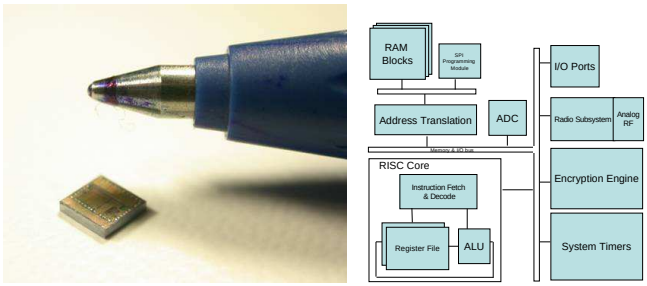
Spec

The Spec node[45] is an ASIC[14] followup to the success of the Mica motes (see Figure 2.6). It continues the design strategy and is based on a single MCU for baseband, MAC and applications with a number of *hardware accelerators* to offload the MCU for demanding operations. To remain compatible with the Mica motes the implemented MCU core is an AVR instruction set compatible, 8 bit, Harvard architecture, RISC core with 16-bit instructions. As the ATMega128l it features a two stage pipeline (instruction fetch/execute) and on chip A/D converter. Additionally a 900 MHz radio transceiver is provided on the chip.

The on-chip radio is a simple device with no offloading features, resulting in high frequency of interrupts for the MCU. To support efficient interrupts two sets of registers are provided (register windows) and an interrupt merely slides the window lowering the overhead of an interrupt to no more than two instructions. Furthermore, a start symbol detection (correlator), simplified direct memory access (DMA) and encryption accelerators are implemented in hardware.

The chip is manufactured in a 0.25 μm technology, measuring 2.5 mm on each side and thousandfold improvements in terms of energy consumption are shown on MCU-intensive operations, compared to the Mica platform.

[14] Application Specific Integrated Circuit



(a) SPEC sensor node (b) SPEC overview

Figure 2.6 *Spec sensor node on a chip. Pictures from Jason Hill's website*[15].

Sensor-Network Asynchronous Processor (SNAP)

SNAP/LE[25] presents an implementation of the SNAP architecture[47], a novel approach to sensor network processors. SNAP distinguishes itself from a general purpose MCU in two ways: it is based on an asynchronous logic and is based on an "event driven" design. The argument for this is the following: recent advances in radio technology will shift the energy bottle-neck such that the energy consumption of the MCU during active instruction execution becomes significant. The event driven architecture will address this problem, while the asynchronous design will provide further energy savings.

Event driven

The processor is based on processing events through an *event queue* instead of signaling interrupt that in turn executing the appropriate interrupt handler. The processor executes the appropriate handlers by removing events from the queue. This eliminates the overhead of handling interrupts. Event handlers are executed non-preemptively, in-order and instructions are issued to the single in-order execution unit. In essence moving the event driven nature of many sensor network applications into the processor.

In addition to the execution unit a timer and message coprocessor is present. The timer unit places events on the queue and notifies the core to execute the proper handler. The message processor is in essence a 16-bit wide FIFO buffer for transmission and reception.

Asynchronous Logic

Clocked (or synchronous) logic uses the clock to determine when signals are *stable* or *valid*—this has two drawbacks in relation to energy consumption: First, when starting the system, the clock-signal has to stabilize, leading to longer startup times and secondly, elaborate measures has to be employed to disable circuitry that is unused in a particular computation (such as dividing the chip into clock domains).

Asynchronous logic eliminates the need for a clock signal by using a *handshake* to express when a signal is valid. This reduces startup times and only transistors needed for a computation will be active, in a sense *automatic* power management.

[15] http://www.jlhlabs.com/jhill_cs/spec

Evaluation

The processor has been simulated extensively using a 0.18 μm technology and a set of tentative power consumption figures are compared to that of the ATMega128l. The authors show an extremely low startup time (in the order of tens of nanoseconds depending on voltage) and a considerably lower energy consumption than the ATMega128l. The expected form factor is not explored.

The comparison does not take into account that the ATMega128l predates SNAP with a few years and is probably manufactured with a greater feature size than the 18 μm of the simulation (the actual feature size is not specified by Atmel). It is unclear what savings can be attributed to the event driven approach, the asynchronous design, or to savings of a lower feature size.

PicoRadio

The PicoRadio Test Bed (or PicoNode I) is the prototype environment of the PicoRadio project. The PicoRadio project is investigating small, low power system on a chip (SoC) devices for sensor networks (or PicoRadio networks). By applying system-level design decisions and meticulous concern for energy reduction they hope to arrive at a much more optimal design than optimizing parts of the system without taking the entire system into account[89].

PicaRadio advocates implementing specialized protocol processors to handle timing sensitive and computing intensive operations. As a first order approximation this can be implemented in a configurable logic block and later refined through a number of iterations to a single ASIC[20]. A substantial energy reduction is observed even with the first order refinement using an FPGA rather than a general purpose micro controller [89].

Pico Radio Test Bed

The Pico Radio Test Bed[14] is divided into a number of PCBs by the logical function: digital (computing), power supply, sensors, radio. They are designed is a stackable fashion for easy and robust assembly. The computing board features (Figure 2.7):

- a StrongARM SA-1100 with 4 MiB RAM and 3 MiB of flash with an adjustable clock from 60 MHz to 200 MHz
- a Xilinx 40 k system gates FPGA[16] (XC4020XLA) with external SRAM and FLASH.

The ARM is running software which will be run on a general purpose microprocessor while the FPGA is emulating the functionality that will eventually be implemented in a dedicated protocol processor.

For the ARM a simple programming environment is provided that provides some operating system services. It is event-driven in the sense that user programs are activated via interrupts (external, timer, etc.) and features a single non-preemptive main routine. The main function is called periodically and it is up to the user to provide parallelism and to ensure that the thread is non blocking.

From the specification above it is clear that this platform is far more powerful than the sensor nodes described previously. While the scenarios and applications envisaged resembles those of most other sensor network projects the described platforms are far more computing intensive resorting to FPGA implementation to solve the computing needs[14].

[16] Field Programmable Gate Array

Figure 2.7 *PicoRadio Test Bed*[17]

While the methodology describes the need for system-level decisions and meticulous concern for power consumption, experimental results using application examples are scarce. Published works describes the completed PicoNode I (Test Bed)[14] and PicoNode II (TCI)[3], but the lack of experimental data is surprising. The axiom of a separate protocol-processor based implementation always out performing a microprocessor based implementation is not shown.

The second approximation of a SoC, the TCI (Two Chip Implementation), is presented consuming 13 mW on average and more than 24 mW peak—and this does not include radio front-end and application processor [3].

Conclusion

Designing SoC platforms allows control over all of the involved components allowing them to be designed for optimal interaction, not being hindered by legacy design choices.

Furthermore, it allows software/hardware co-design—the SPEC node included an encryption engine, the SNAP processor moved the event based execution into the processor. In this way the designers are able to build the exact features that are required in a given application, this would have been impossible using COTS components.

2.2.4 Discussion

In this section we described the strategies for building platforms seen previously in the sensor network community. We have looked at generic nodes built on a set of assumptions about common general purpose sensor network application, using commercially available components. Finally we looked at the early stages of two SoC sensor nodes.

Most of the platforms we described have never been tested in long deployments or large numbers. It is our claim that the generic sensor nodes focus on the *functional* aspects of a sensor network application while the *performance* is non-optimal. Based on the few deployments we have seen we are convinced that such platforms will not be able to achieve the performance required by Hogthrob. Therefore we do not use any of the exiting nodes. We need a sensor node specialized to our conditions. It must be cheap (a few €), small (can be attached to the ear of a sow), have a lifetime of up to two years.

[17] http://bwrc.eecs.berkeley.edu/Research/Pico_Radio/Default.htm

We will not go into the cost of producing electronics, but it is a given that for high volumes integrated circuits (chips) are much cheaper than mounting similar components on a PCB. As an example a singe Mica2 from Crossbow costs 150 USD[18] and this does not include sensors while the ATMega128l integrated MCU with numerous peripherals costs about 10 USD. It is our claim that to be able to build a sensor node including sensors, microprocessor and radio withing the budget it must be built as a system on a chip.

The previous deployment examples showed us the importance of taking all layers into account when designing a sensor node. Component boundaries impose a limitation on the type of functionality that can be implemented. The Great Duck Island did not nearly meet their lifetime goals, Zebranet were constrained in the power-saving features the could utilize by poorly integrated components (a similar observation was made for TinyBT[61]).

To solve these problems we choose to follow a *holistic* view that takes all layers of design into account when constructing a sensor network. We do this by employing hardware and software co-design[53]. That is, we need to design and evaluate the ability of the hardware platform to support the application and we need to design and evaluate the ability of the software to exploit the power conserving features of the platform as well as supporting the needs of the application.

The question is now: how do we design and evaluate a SoC? Producing chips takes time and is expensive, consequently we wish to evaluate the ability of the SoC design to meet the application requirements as a part of the design process—before a SoC is available.

Lifetime being the most important parameter, how do we estimate the lifetime of such a sensor node? We will return to this topic in Chapter 3.

2.3 Sensor Network Software

As in many other areas the rise of sensor networks has challenged some of the conventional wisdom within software systems design. This has resulted in exploration with a large number of areas, encompassing larger or smaller parts of the ecosystem relating to a sensor network deployment. In the following we will look at some of the main trends and discuss examples. The software support systems are in charge of acquiring the data and reacting to the results. This could involve tasks such transporting, actuating, aggregating, etc. To accomplish this task sensor network designers structure applications to their needs and the requirements of a particular example.

Today most systems employ a tiered approach implementing some functionality at a level above the motes. Motes run embedded software that sample the physical environment off loading data, events, aggregates or similar to a second tier. This is a contrary to the initial belief in the sensor network community, that most processing would be pushed to the network[39].

Regardless of the tiered approach each mote still needs a program, and the most common approaches to programming each mote, is to either program it using some form of operating system or to choose a higher level of abstraction.

2.3.1 Mote Operating Systems

Motes run the software that sample the physical environment and communicate with peers or gateways. Quite traditionally the motes are usually programmed using an operating system, regardless of how the upper layers are designed. The operating systems however vary from

[18] From the Crossbow website, December 2007 http://www.xbow.com

traditional operating systems in terms of goals and techniques. Consider for example dynamic loading of programs. On a PC size operating system this is essential. On a mote simply replacing an entire image with a new one may be sufficient.

Most operating systems abstract the underlying hardware, but each system differ substantially in the approach to memory protection, dynamic reprogramming, thread model, real-time features, etc. The mote operating systems have grown in parallel with traditional embedded operating systems, and most sensor network project focus on other areas than operating systems from the embedded arena.

High Level Abstraction

One trend is to reduce complexity of the sensor motes is to abstract the functionality at a high level. In this way programming the entire network is significantly reduced. For example by exporting a set of common functionality blocks configured at run time (Tenet, TinyDB, Arch Rock, Sensorware), by providing a virtual machine that is re programmed (Maté, Sunspot, Sensilla)

a different approach, that we have not discussed is the use of virtual machines, to this date this approach has not caught widespread popularity in the research community. Recently, however, the company Setilla[19] moved in this direction by providing low power, highly efficient sensor motes based on a Java virtual machine. This move ensures portability, and simultaneously opens the sensor network domain to a world of Java programmers.

Direct Programming

The operating systems supporting direct programming of the sensor network motes differ substantially in design. The focus of these projects is to provide features that are closely matched to the application area of sensor networks: low overhead, constricted memories, diverse hardware platforms.

The majority of project draw from existing experience and implement subsets of features that are developed in other areas. Adopting a *kernel* that abstract hardware features is a common approach. The approach to memory protection, multithreading, scheduling, reprogramming, etc. vary substantially.

The event driven nature of sensor network applications has lead to the concept of event driven operating systems[43]. The event driven concept relies on executing a thread as result of an event, rather as a periodic scheduling. This strategy eliminates the need for a per thread stack and some or all of the context switching overhead. TinyOS, SoS[37] and Contiki[23] takes advantage of this observation.

Contiki extends this model with the thread library ProtoThreads[24] providing more traditional preemptive thread scheduling. We will describe TinyOS in detail in a moment, but the event driven model is extended with more traditional threads through the TinyThread library[73]. In addition SoS and Contiki focus on dynamic program loading and dynamic memory allocation. SoS further focuses on the composition of programs by encapsulating programs into modules with a messaging and function interface to the surroundings.

A more traditional approach could be to attempt to provide some of the features commonly found in PC-size operating systems. This substantially reduces the learning curve for programming sensor networks, and provides some of the same benefit known from full fledged operating systems. The current strategy seems to focus on providing a subset of common features such as memory protection, preemptive scheduling, realtime functionality, dynamic memory

[19] http://www.sentilla.com

allocation. Such an approach is taken by Mantis, BTNut, t-kernel, Nano-RK, FreeRTOS, LiteOS, μC/OS, ARVX

2.3.2 TinyOS

By far the most popular operating system within recent sensor network projects is TinyOS. TinyOS based on two observations regarding sensor network application: they are event driven and support multiple streams of data. TinyOS programs are driven by a event/command interface, supporting fast event executing. An event may delay execution by posting a task, a task corresponds roughly to a thread, but are executed to completion and one task cannot preempt another task. This mechanism provides parallelism by cooperative scheduling. The simple nature of TinyOS makes it very light and independent of hardware features of a particular platform.

TinyOS built around a component concept, and a TinyOS program is a composition of new and existing programs. Each component use and provide a set of interfaces that can be connected with their counterpart from other components. Components and interfaces are written in the C-extension nesC that provides the semantics for interface definitions and component assembly. The clean interfaces creates a clear driver/application boundary that is idea for abstracting hardware differences. In this way TinyOS is just as much a programming environment, as it is an operating system.

TinyOS compiles all source components to a single C file that is compiled using a C compiler. This strategy allows a number of compile time checks that would otherwise have been difficult to archive. Whole program analysis by the compiler, optimized inlining, interface contracts, compile time stack analysis, compile time memory safety, interface contract are some of the recent examples.

One of the drawbacks of TinyOS is the learning curve, by departing from the programming paradigm taught to every computer scientist on the planet the starting. Apart from taking some getting used to it has been argued that the programming model is much more difficult to grasp and makes simple tasks more difficult to implement[24, 73], on the other hand relying on threads to solve difficult problems can lead to erroneous behavior[56].

The recent TinyOS 2 version provides a set of documents describing the abstract functionality that each platform must provide as a set of interfaces. Each platform in TinyOS is required to provide these interfaces.

Further Reading

While the topic of TinyOS is relevant, an in depth discussion of TinyOS is beyond the scope of this dissertation. We direct the interested reader to some of the following sources for further information:

- A recent overview paper of TinyOS is available here[32] and the original design considerations are available here[43, 45]. Further information is available through the TinyOS website:

  ```
  http://www.tinyos.net
  ```

- An excellent programming guide is available here [62].
- An overview of the Hardware Abstraction Architecture (HAA) used as the foundation of TinyOS 2 is available here [38].

2.4 Power Estimation in Sensor Networks

While it is generally accepted in the sensor network community that energy consumption is the crucial evaluation metric, the amount of work on estimating power consumption is surprisingly low. Most studies have centered around optimizing specific subsystems of a sensor node most importantly communication—only few look into the power consumption of entire networks, or evaluate the performance of actual deployments.

Until now we have established that we need to build a SoC. To assist us designing the SoC we need a methodology to evaluate the design decisions in the context of our application. This brings two design disciplines together: sensor network design and VLSI design. In a sensor network capturing the behavior of the application also implies capturing the behavior of all layers of the node: sensors, networking, operating system, etc. Power estimation in the context of digital design rarely go as high as the operating, let alone the network and the surroundings.

We will begin by discussing two strategies for power estimation of sensor networks applications (Section 2.4.1). And go on to describe the related work in the context of sensor networks (Section 2.4.2) and VLSI design (Section 2.4.3).

With basis in previous work we will construct a taxonomy that will be used to construct a model of power consumption for the Hogthrob project (Section 2.4.4). Finally we describe the strategy for the Hogthrob project (Section 2.4.5).

2.4.1 Power Estimation Strategies

The two power estimation strategies that have been employed in the sensor network community on rely either on direct measurement or simulation. The two strategies differ primarily (a) in the way the program is executed and (b) in the way inputs are given to it.

Direct Measurement

One way to gain insights as to the power consumption of a given sensor network is to deploy it and measure the performance. Data can be collected either on-line or stored for *post mortem* analysis. This relies on using an instrument to measure properties of a sensor node running the program binary, while the surroundings is stimulating the inputs of the sensor node. The recorded log or trace can consist of either measurable values (current, voltage, etc.) or indirect measurements such as the number of packets, I/O activity, etc.

During a field experiment the inputs of the sensor node are stimuli from the environment and radio communication with other nodes. We call these inputs *real* as opposed to *synthetic* inputs emulated by a model during simulation or lab experiments.

Notice that we distinguish lab experiments from field experiments—lab experiments will in general not recreate the environment that we are trying to observe. Sensor network literature show us that field experiments often yield surprises compared to lab experiments.

The Great Duck Island expedition relied on direct measurement of the sensor nodes, but encountered several cases of behavior in the field deviating from the expectations (based on lab experiments). We are convinced that these deviations show that the lab experiments did not turn out to be representative of field experiments and did not provide the authors with the insights about the performance in the field, that they were trying to find.

Deploying and measuring on large numbers of nodes is difficult: the nodes may not be available or practical challenge of deploying nodes may be unattractive. A short-cut to these

problems is provided by simulation.

Simulation

A software simulation of a sensor network can be carried out *a priori*: a model of the sensor node and software is simulated while stimulating it with synthetic inputs. During the simulation run power relevant information is recorded. The nature of the information is entirely up to the simulator and can be as detailed or as abstract as required.

Common techniques for generating inputs for the simulation range from synthetic environment models to replaying captured traces of previous measurements (often network traffic). While great care can be taken when constructing environmental models they remain synthetic and only model the world as the designers believe it to be.

PowerTOSSIM is a novel approach to power estimation using simulation (see Section 2.4.2). PowerTOSSIM derives the execution model from the program binary, but relies on simulated inputs. While PowerTOSSIM estimates the power consumption with low error on a number of examples it shows high error (above 10%) on two crucial benchmarks: a beacon operation using the low-power states of the MCU and a light sensing application using the distributed query system TinyDB[70]. The authors spend little time investigating the causes of these errors but merely suggest that they could be caused by "*inaccuracies in the (MCU) cycle count*"[94, p. 9] and "*partly due to the fact that TinyDB exhibits somewhat different behavior in simulation than it does in actual hardware*"[94, p. 10].

Discussion

The program and input together determine the behavior of the sensor node and are therefore essential to the power estimation process. The two techniques described above differ in their approach to these two subjects giving rise to different problems.

Direct measurements presents an exact execution model and it allows real inputs. Viewed in isolation each sensor node exhibits high determinism—it samples measurements based on a timer and forwards them to a base station. Non-determinism is introduced in the interaction with the environment and other nodes. Capturing this dynamic behavior and the impact on the application using only a software simulator is difficult. To capture the impact of this dynamic interaction field experiments are required.

On the other hand using software simulation is helpful to get the big picture or for use in the early stages of a project. Deploying and measuring on a great number of nodes is, however, in itself a daunting task and the required instrumentation itself can disrupt the measurements, by polluting network traffic or consuming energy. The major drawback of a simulation, however, is the dependence on the model of the sensor node and the environment—imprecision in these models can produce incorrect results.

In the context of Hogthrob direct measurements are not possible since our SoC is not available. This means that not only will we have to estimate the impact of the surroundings we will also have to estimate the power consumption of the hardware.

2.4.2 Node and Network Level Power Estimation

Estimating power consumption in the context of a sensor network involves simulating the network and the sensor node. Distinguishing between *node level* and *network level* simulators can be advantageous as the techniques to model either differ substantially. Most network level simulators do not accurately model the node and vice versa, making it hard to make power estimations for an entire network using one or the other[79].

Simulating sensor nodes has many similarities to simulating embedded hardware and we start out by describing simulation environments originating from embedded hardware and return to sensor networks in Section 2.4.2.

Embedded Systems

The embedded systems community has produced a diverse number of strategies for power estimation originating in hardware design. A number of academic and industrial power estimation tools with emphasis on viewing the system as a whole have emerged, most noticeably a number of *architecture level simulators*, simulating functionality processors at a high level. The models are often limited to the processor cores, and in the context of sensor networks they disregard other components such as sensors and radios.

SimpleScalar[4] models the functionality of each internal processor block, and is able to simulate the exact behavior of each pipeline step, cache-block, data register, etc. This model is augmented by the Wattch[13], SimplePower[104], and $\mathrm{TEM^2P^2EST}$[21] with different power consumption models. A set of reusable hardware components models and uses SimpleScalar to track the usage patterns of these blocks in each cycle. The models available for SimpleScalar focus on much more high performance processors than the ones seen in sensor networks, such as pipe lined processors with large memory caches.

AccuPower[86] present a reimplementation of SimpleScalar that increases precision of the simulated results, but is based on the same principles. They implement critical parts of the processor model in a HDL description language and perform detailed, analog simulation on these parts. This technique is in their own words *short of an actual implementation, AccuPower's power estimation strategy ... is as accurate as it gets*[86, p. 2].

JouleTrack[96] explores per-instruction (or *instruction level*) energy consumption of two high performance embedded processors (Strong ARM SA-1100 and Hitachi SH-4). They observe that the energy consumption by instruction type is largely dominated by a common overhead (decode logic, caches, etc.) and as a first order approximation can be regarded as equal. A second order approximation is proposed grouping instructions into classes of power consumption. A power estimate of a program is computed by collecting instruction statistics of a program and feeding them into a model containing the two observations.

A different approach to simulating a detailed model consists in measuring the performance of the program running on real hardware and relate this to the program source. While this technique is appealing, doing this in practice requires some work.

PowerScope[26] compiles a per process power profile by having an external PC regularly sample program ID and current draw. SES[93] presents an add-on real time capture card that can not only take exact power measurements, but correlate these to the exact instructions of the program. EmSim[99] obtains a similar correlation of measurements and program execution by simply raising an I/O pin.

Sensor Networks

In the sensor network community, we boil the approaches down to two techniques for building simulation environments: the one continues the thread of *instruction level* simulation, simulating the exact execution of the sensor node binary. The major drawback to this approach is scalability. This assertion has recently been challenged by the AVRORA[20], but no publications are available at the time of writing. The other simulates a functionally equivalent of the sensor

[20] http://compilers.cs.ucla.edu/avrora

node software. The most established example of this approach is TOSSIM, that does not provide power simulation[63].

Instruction level simulators are presented in EmSim[99], ATEMU[85] and by Robert Dick[22]. The behavioral implementation of each functional unit is augmented with power estimates from literature (data sheets, etc.) or experiments and power estimates are computed using usage statistics (following the black-box view of components of [95]).

SensorSim[80] presents a sensor node and network simulation environment. The application is modeled in the TCL-based SensorWare[12] execution environment. The networking model builds upon the ns-2[21] model of Wireless LAN (802.11) and the notion of *sensor channel* models the environmental stimuli flowing to the sensor nodes. In addition to the TCL functional description, SensorSim includes a power model in which each platform component (MCU, radio, etc.) report power state changes to a power source, and the drain is computed. The framework allows an individual power model for each device, and a model for the MCU and radio is described: The radio tracks changes in operation mode (i.e. receive, transmit, off, etc.) and a rough cycle count is assigned to each task in the simulated program, assuming equal cost for all instructions.

SensorSim recognizes the difficulty of stimulating a simulation with realistic models of the environment and allows a simulated node to be connected to the surrounding world by a real wireless interface.

ESyPS[79] emphasizes the node/network level simulator by combining the network simulation properties of SensorSim and extend Princeton EmSim[99] with a sensor model. Each node in the simulation is either a SensorSim node or an ESyPS node, and the two simulations are synchronized. In this way the feature to be emphasized is selected to for each node.

EmStar recognizes the difficulty of producing realistic stimuli for a simulation and proposes a hybrid mode: simulated nodes are connected to the outside world with real wireless connections. EmStar focuses on heterogeneous systems by providing system services for interconnecting a mix of sensor network nodes, more powerful micro-servers and PCs. Furthermore, it is able to simulate the execution of each of these devices[33].

An alternative approach for extracting a model of the behavior of the software is to model the TinyOS component graph as a *hybrid automata*[22]—a high level platform independent representation for both correctness analysis and power estimation[17]. The execution of event handlers is modeled by states accounting for the number of clock cycles to execute an event and the time spent waiting for events. By tracing the flow of this model, a power consumption estimate is computed.

SENS[97] emphasizes on the environmental impact on sensor network simulations and provides simulation in a fashion similar to TOSSIM. An, API[23], allows easy integration with for example TinyOS programs that can be compiled, and executed on a work-station. It provides a power model much in the style of SensorSim—the application and networking components report relevant power transitions to a central entity that handles bookkeeping. It models the interaction with the environment in a similar way to the SensorSim sensor channel.

PowerTOSSIM

PowerTOSSIM is an extension of TOSSIM and leverages the scalability of TOSSIM, but enhances the simulation with power consumption estimates. The cost of the scalability of PowerTOSSIM and TOSSIM is precision, PowerTOSSIM and TOSSIM scales to thousands of nodes easily on a

[21] The Network Simulator http://www.isi.edu/nsnam/ns

[22] a mathematical model capable of describing both discrete and continuous behavior

[23] Application Program Interface

desktop PC while more precise, computing intensive techniques would require more time and computing facilities.

PowerTOSSIM simulates an application using TOSSIM and captures a trace of power relevant transitions. This trace is fed to a *power profile* and using the timing information in the trace and the information in the profile a power consumption estimate is computed. Such a profile details the power consumption of the individual components CPU, radio, sensors, LEDs, etc. The authors construct a profile for the Mica2 platform using a number of synthetic benchmark applications that each exercise certain components of the platform.

The scalability of TOSSIM stems from the fact that the applications are compiled as native executables for the simulating platform—it is not simulating the actual instructions on the target platform. In order to estimate the power consumption of the MCU on the target platform the authors impose a novel code transformation technique that relates the target code to the one on a PC. By counting the number of *basic blocks* (sequences of instructions without branches) in the simulation binary and relating this to the corresponding block in the binary for a sensor node, an estimate cycle count is obtained. Experiments show that this technique has acceptable precision for common applications.

PowerTOSSIM addresses the power consumption at the application level. Using this framework design choices at every level can be simulate by either changing the application or the power profile.

Discussion

To sum up we saw a diversity of strategies for estimating the node level and network level issues. Each simulator presents a model that is calibrated to a known truth, putting emphasis on particular issues at a given level of abstraction. In general, two effects are disregarded:

Fixed voltage the power consumption of electrical components varies with voltage[24]

Instant Startup most electrical components have a non-zero startup time. For sensor network systems that frequently power on and off this startup time is significant.

The majority of the examples above build their models upon existing hardware. In our case the hardware is not available—we need to design and estimate the power consumption of non existing hardware.

2.4.3 VLSI Design

A major part of the SoC we are designing is the processor and associated hardware accelerators. Such digital hardware components are commonly described using some form of High-level Description Language (HDL) such as VHDL, Verilog, or SystemC. It is this high level description that is eventually implemented in a chip (often denoted as Application Specific Integrated Circuitry or ASIC).

Building and simulating digital hardware is an engineering discipline of its own. While it is not the topic of this thesis we need to give a short background to discuss the available power estimation techniques. We will outline how the digital design process takes place in Section 2.4.3 and return to the related work on estimating power consumption in Section 2.4.3.

[24] A CMOS circuit can as a first order approximation be considered as an ohmic resistor in witch case the power consumption P can be described as $P = \frac{U^2}{R}$

When looking at the design process, note that the design methodology of the Hogthrob project is mirrored within the design levels of the HDL design process. Making a power conscious behavioral decision has the potential to yield much higher power savings than making an efficient implementation in transistors, just as making an application level decision has much greater potential then trying to optimize a flawed design.

Digital Design Flow

It is beyond the scope of this work to go into the details of the design and evaluation of a HDL model. However, in order to discuss the power consumption simulation techniques, we will briefly summarize the design flow from a high level HDL model to an actual ASIC implementation. The common approach to design and implement a desired behavior is evolve the design through a number of steps or levels . At each step the design is gradually refined and decomposed into modules that will eventually be implemented in hardware components. Transforming a functional description of the desired circuit to an implementation in logical gates and finally an implementation in transistors.

The refinement process is semi automated and assisted by advanced compilers (or synthesizers) and the functionality can be simulated and compared to the model or simulation of a different level (for example gate level versus, register level). Each of these simulations can be augmented with a model of the power consumption of a given implementation and there by giving estimates. The precision of such models increases as we approach the lower levels simulating the physics of the circuitry. Such simulations are extremely computing intensive and hence time consuming.

Even at the higher abstraction levels software simulation is computing intensive and time consuming. As an alternative the design can be simulated using a hardware simulation in the form of a reconfigurable logic block. Such a device has a number of generic logic blocks that can be rearranged to match any functionality. The logic block manufacturer provides tools that synthesizes the design not to a implementation, but to a configuration of the logic block that matches the functionality of the design.

Estimating Power Consumption of a Digital Design

The power consumption of a circuit consists roughly of two parts: the *static* power consumption and the *dynamic* power consumption[15, 30]. The static part denotes the base-line leakage that the circuit exhibits, while the dynamic is derived from the power consumed by transistors during logic transitions (the switching activity). The power consumption of a design in a given application is therefore a factor of the chip layout and the actual inputs resulting in a specific switching activity.

Estimating the power consumption of digital designs has been recognized as a first-class design constraint not only in mobile computing applications, but in many other applications ranging from multi media through high-speed networking devices to super scalar microprocessors [30, 86]. Consequently a number of techniques exists that can assist us in the process:

HDL tool chain commercially available HDL compilers are able to give estimates of expected power consumption at different levels of design.

Rules of thumb comparing the design to other designs might give just as valuable information as a HDL compiler.

[24] Alliance is available for download at http://wwwasim.lip6.fr/recherche/alliance

Hardware simulation simulating the design in hardware can give us valuable information allowing us to estimate power consumption.

HDL tool chain

At the lowest level a design can be simulated using an analog simulator often denoted as a SPICE simulation after the most well-known simulator (see Section 2.4.3). Such a simulation is only possible at the point in the design process when an analog representation of the design is available and is its time consuming. At higher levels of abstraction the simulation time decreases while the number of unknowns in the modeling increases. For example, simulating processor cores at the architecture level (RTL) level is troublesome, as many of the implementation details that affect power consumption greatly, has yet to be determined.

The simulated results produced by the different steps of the HDL tool chain relate to the power consumption that can be expected in the implementation not only in the precision of the model, but also on a number of other factors.

First the compiler uses a library of common components (cells) to generate the layout of the chip. This library contains implementations of different types of transistors, gates and other functional blocks. Chip producers often supply or compile designs using their own library optimized to the production plant. Each plant often supplies these libraries describing the performance that can be expected, but most research projects do not consider a particular chip manufacturer.

Secondly the exact performance of a given gate, transistor and more differs slightly from production run to production run. The variations are often controlled as a contractual matter, but in some cases a production run can vary slightly more from the mean than another.

The large number of unknowns before the chip is actually produced means that the values must be viewed with some skepticism.

Rules of Thumb

Producing a rough estimate using common sense can in this context produce sufficient precision. One could argue that the large number of unknown factors that can radically change the outcome of the chip production reduces this to little more than automated rules of thumb. Along these lines, performing simple calculations can give us a rough estimate.

Simply designing a spreadsheet with basic power consumption figures and filling out the blanks from the output of a HDL compiler will in many cases suffice. For example the Xilinx Power Tools[25] contain a spreadsheet[26] and web edition[27] to estimate the power consumption of an FPGA design.

Hardware Simulation

While a reconfigurable logic block is functionally equivalent of its chip-implementation counterpart, the energy consumption is different in a number of ways:

- The baseline power consumption is factors higher than that of a corresponding chip.
- The mapping (synthesis) of the design onto either on to the generic logic or to an ASIC implementation radically different. As a result the *dynamic* power consumption will differ.

[25] http://www.xilinx.com/products/design_resources/design_tool/grouping/power_tools.htm
[26] http://www.xilinx.com/ise/power_tools/license_spartan2e.htm
[27] http://www.xilinx.com/cgi-bin/power_tool/power_Spartan3

This means that a direct measurement of the configurable logic block does not translate into either *relative* nor *absolute* power consumption of the SoC with the same functionality.

To estimate the power consumption using a reconfigurable logic block a *mapping* is required. A number of options exist to perform such a mapping. One option is to carefully investigate the synthesis to the logic block configuration an the synthesis to the ASIC implementation and relate the two to each other.

Another option is to use *on-chip analysis*. By implementing an interface in the logic block that allows us to monitor the activity inside the chip, this activity can be mapped to the performance of the SoC.

VLSI Power Estimation

When estimating the power consumption in a digital design project, this usually takes place quite late in the process; consequently most of these tools try to raise the abstraction level at which the power simulation takes place.

Both commercially and academically, the subject of power estimation has received great attention within the VLSI[28] domain. Traditional simulation of hardware designs are performed using analog models of the circuit, the Berkeley SPICE or BSIM[29] are examples of academic tools. Such simulations are computing intensive and increasing the speed of the estimation while retaining good precision is a prime issue[88].

A number of techniques have been suggested to speed up this simulation, in essence attempting to elevate the point in the design process at which we are able to perform power estimation. This subject is still a topic of research and we will not go into the details, but only mention a few.

High level power estimation is often synonym with RTL level simulation. A number of techniques have been developed both academically and commercially that are able to estimate power consumption given an RTL level description of the circuit. Among these techniques are[90]: *macro-modeling* and *fast synthesis*.

Macro modeling is a statistical method that attempts to characterize a lower level implementation of various RTL macro blocks by means of statistically "training" the model with random inputs. Typically, a gate or transistor level tool is used to evaluate the power consumption of a block given each input. HyPE[65], [87], [90] are academic examples of this technique.

Fast synthesis is a technique by which to short-cut the synthesis process resulting in an approximate design. The process is simplified by providing a library of larger functional blocks that is used to map the RTL description to a design. This allows the designer to get an early view of the physical effects of a design without resorting to a full synthesis, that can be time consuming and troublesome. Sequence PowerTheater[30], Atrenta SpyGlass[31] and Terasystems TeraForm[32] are commercial examples of such tools.

In the Hogthrob project we rely on the Synopsis Power Compiler[33] augmenting a RTL level functional simulation with physical information from a library for a given chip technology[50].

[28] Very-large-scale integration (of transistor based circuits)
[29] http://www-device.eecs.berkeley.edu/ bsim3/
[30] http://www.sequencedesign.com
[31] http://www.atrenta.com
[32] http://www.terasystems.com
[33] http://www.synopsys.com

2.4.4 Power Model

With the overview in power estimation techniques originating from sensor networks, embedded systems and VLSI design we are now armed with the tools to abstract and build a set of concepts that we will use to define a power estimation technique for our system on a chip.

Consider how power is consumed on a sensor node: energy flows from the battery trough the electrical components in the form of electrons—we are trying to capture the amount of energy at any given time and the total amount of energy dissipated in the circuit. Note that the exact nature and time dependence of the power consumption for each component is complicated and is influences by parameters such as temperature, the chip technology, the particular production run[34].

We wish to abstract the details of power consumption while retaining sufficient precision. To do so, we define a collection of states and transitions between them, for each component in the system. To each state or transition a given power consumption cost is associated. Some costs are a function of time (e.g. radio on, power-down) while others are fixed (e.g. the execution of and add instruction, or the switching of a transistor). The states represents differences in power consumption at different times, for example switching from an active to a low-power mode of operation. A collection of states is what we call a *power model* and the associated costs we call a *power profile*.

Choosing the *granularity* of states is entirely dependent on the required precision and on the component in question. A sensor node is made up of a number of components with different characteristics, it is essential that the power model captures the nature of each of these components. For the most part when working with commercial components it is usually not possible to view the intricate details of what goes on inside, thus imposing a *black-box* view on the component[35].

For example, consider some of the power models of commercial MCUs described previously. While very few details are available, good results are obtained by employing a wide spread of strategies ranging from guessing the probable function blocks, counting groups of similar instructions, counting instructions to counting basic blocks. The previous work shows us that when working with commercial components the model of the component, is often constructed based on the circumstances rather than choice.

The power model is not a quantitative measure. It only describes *how* the components of the system consume energy. As such it does not in itself provide an estimate of the power consumption of an application running on the platform to the application running on a SoC. In order to do this we need to:

- Map the sensor node execution to the states in the power model, we do this by capturing a *trace* of the activities.
- Map the states of the power profile to actual power consumption figures. We do this by accompanying the power model with a suite of measurements representing the SoC power consumption—the *power profile*.

Let us proceed to discus the traces and the power profile.

[34] The energy consumption is described by the Joule heating law expressing the power P dissipated a steady current I in the electric potential V as $P = IV$. In our case the current and voltage is time dependent and we express the total amount of energy as $\int V(t)I(t)dt$[35].

[35] Abstracting components into black boxes in this way works well for digital designs, but for analog designs it is often insufficient. When designing an analog amplifier, chip microphone or some similar low-power analog component the precise analog performance of the chip will determine the performance. A common technique is to employ analog simulations using models of the production technology provided by the chip manufacturer.

Trace

In order to estimate the power consumption of the SoC we will have to gather the relevant information that will allow us to abstract from the details of the prototype platform; we define this information as the *trace*.

The subsystems commonly seen on sensor nodes are: computing, communication, sensing and power. The components associated with each of these subsystems have different characteristics and a number of different techniques have been employed to capture an activity trace of each:

Computing Capturing the activity of the computing subsystem is commonly done by recording instruction or a statistics processor of function block usage.

Sensing Sensors can be entire subsystems, but the functionality is often limited and easily abstracted: turn on, sense, turn off. However, actual components might not be as simple—some sensors might not provide sufficient low-power modes to turn completely off or consume power continuously whether sensing or not. Most importantly, most components have a startup time, during which the power consumption grows to its final level—the duration and slope of this is significant.

Communication Capturing the power consumption of the networking devices range from tracking the states of the radio to capturing traces of the networking traffic itself[36].

Power The power subsystem have in general been disregarded in the traces we have seen. The power subsystem is often composed of a number of DC-DC converters that have significant power consumption (loss) and different characteristics of other components. We return to this subject in Chapter 3.

Traces in situ

A number of problems emerges when trying to gather data from nodes deployed in the field rather than in a lab experiment or even in a simulation.

- The amount of data produced by the trace can be quite large. We have not discussed the type of possible traces, but consider attempting to collect an instruction trace for an instruction level model. Capturing a snapshot of the currently executing instruction at every clock cycle of a simple 7 MHz processor with 16 bit instructions means capturing a stream of 112 Mbps.
- In most cases the required instrumentation will in one way or an other affect an experiment for example by generating artificial network traffic or by reducing node lifetime. Tracking and calibrating this influence is essential in order to compensate the observations.

Gathering Traces

We imagine three strategies for gathering the traces in situ: Storing the data on the node for post mortem analysis, forwarding the trace via the usual data channel or using a designated debug channel. All have drawbacks and selecting the best one depends entirely on the circumstances.

[36] Capturing network traffic is very high-level, indirect approach. From these measures the details of the will have to be reconstructed and it is therefore associated with great imprecision

Storing the data on the node for post mortem analysis requires space and energy. While this approach is unable to supply on-line access to debug information, such data can be very useful when investigating node behavior or failure. Even conducting shortened experiments in order to gather data about the sensor network itself might be attractive. The authors of the Great Duck Island experiment suggest that this cost might be worthwhile[98].

Forwarding the data via the communications channel will of course generate artificial network traffic disturbing the sensor data traffic. While this influence could be minimized using techniques like piggy-backing, aggregation or compression, if it is the networking characteristics that are being monitored, this is not an attractive solution. The EmStar[33] framework accumulates data in a buffer and forwards it in a space-efficient, compressed, binary format.

Having a wired or wireless back-channel will for many sensor network deployments be impossible or infeasible, but for lab experiments where power and fixtures are close by this seems like the most attractive solution. While the back-channel might seem zero intrusive, bear in mind that unless the monitoring is completely separate from the sensor node, it would still have to devote time and energy to sending data via the channel.

Detecting Events

While detecting event is easy and free in a simulation doing it in the wild either requires the software to be instrumented with debugging facilities or that elaborate "snooping" features are available. Most sensor nodes do not have such won board snooping facilities and will have to rely on either instrumenting the code (such as the Great Duck Island[98] and EmStar project [33]) or designing a separate monitoring board[37].

Instrumenting the code will impose some overhead in terms program execution and energy consumption, but without any facilities, this is the only way to gain knowledge of the program execution.

Power Profile

The topic of assigning an energy cost to a specific state has been investigated mostly in the context of simulation. In most cases, this relies on *calibrating* a model to some known truth using either *simulation* or *measurements*. Often the details are abstracted by computing *average* power consumption over a certain period of time, during which the power consumption is can be considered constant. Whether measurements or simulation is used rely on which type of model is available:

Measurement The majority of the models are calibrated with data from commercially available components. A common approach is to exercise particular features of the component using micro-benchmarks and measure the performance. The level of detail of such benchmarks varies substantially, from measurements of instruction level costs, to viewing a PC as one unit of PowerScope. In general, the models we have seen are limited by the level of detail that can be obtained from component manufacturers.

Simulation If a more detailed model is available, this model can be simulated. SNAP was able to do this as the source of the processor is available, but Wattch uses a similar technique by simulating *generic* subcomponents commonly seen in microprocessors.

The techniques described previously cover both techniques and good results have been obtained by calibrating even fairly imprecise models with measurements of micro benchmarks.

[37] Such as the XBow MIB600 ethernet-connected programming board used in the MoteLab project http://motelab.eecs.harvard.edu

2.4.5 Discussion

The techniques we have covered until now rely on two assumptions that differ from the assumptions in the Hogthrob project: direct measurement rely on existing hardware and simulation using synthetic input. Both are valid within their domain, but in the Hogthrob project direct measurements are not possible since our SoC is not available, and in order to impose an application driven design we need real inputs.

As a consequence we need a third approach combining the dynamic behavior of real experiments with the details of a low-level simulation.

The approach that we propose is to rely on a hardware simulation of the sensor node design. We do this by implementing a prototype platform that allows us to change every aspect of the sensor node design, including sensor, radios, but most importantly it allows us to explore the benefits of implementing microprocessor features specifically for our application. Simulating such changes is achieved by employing *reconfigurable hardware*. This platform is just as flexible as the software simulation and allows us to deploy the simulation in the field.

The prototype node will be *functionally* equivalent to the system on a chip, while we need a power model to map the *performance* to system on a chip. The techniques we have covered in this section attempted to model the behavior of existing platforms and existing components. In the context of Hogthrob we are trying to do more than model a node with a MCU connected to a set of components; we are trying to move the entire system onto one chip. This complicates the calibration as we are dissolving the component boundaries seen on sensor node platforms.

In Chapter 3 we return to the subject and describe how to capture a platform independent trace of an application.

2.5 Hogthrob Prototype Platform

The Hogthrob prototype platform or HogthrobV0 was the first approach taken by the Hogthrob project. The idea is to use this platform as a general purpose testing device throughout the project. The platform is manufactured by a private company and designed by the by the project partners collectively:

- Dept. of Computer Science (DIKU), Faculty of Science, University of Copenhagen (KU)
- Dept. of Large Animal Science, Faculty of Life Science, University of Copenhagen (LIFE)
- Informatics and Mathematical Modeling (IMM), Technical University of Denmark (DTU)
- The consortium also consists of the National Committee for Pig Production[38]. In the beginning of the project IO Technologies[39] assisted in the construction of our prototype development platform.

The Hogthrob prototype platform (HogthrobV0) must serve as a development platform throughout the Hogthrob project. It must be general enough to allow a large variety of configurations and robust enough to allow lab and field experiments. The design goals of the Hogthrob prototype platform are different from that of the sensor node we are trying to build. It must be functionally equivalent of our sensor node on a chip, and we must be able map the design to the performance of a sensor node on a chip.

[38] http://www.landsudvalgetforsvin.dk
[39] Now Prevas http://www.prevas.dk

The platform must be flexible enough to let us change any of the givens of the sensor node design: radio, sensors, microprocessor, hardware accelerators, etc. This allows us to explore a broad spectrum of design choices: hardware/software boundary, radio protocol design, duty cycling, sensor sampling frequencies, etc.

The two major goal of HogthrobV0 are

- to allow software/hardware co-design
- to provide a prototype platform for further exploration of the design space.

To achieve these objectives we adapt a modular design strategy so that we can swap sensors or radio transceivers with ones resulting in more efficient energy and system performance. To experiment with microprocessor designs and/or hardware accelerators, we need some form of reconfigurable logic on the prototype platform. We choose to implement these goals using a Xilinx Spartan III FPGA with external FLASH for the FPGA configuration and for the program running on the FPGA. In addition we placed an ATMega 182l MCU that provides A/D as well as housekeeping for the FPGA power up/down procedure.

2.5.1 HogthrobV0

The platform was defined by the Hogthrob partners and was implemented by I/O Technologies delivering practical expertise in embedded systems design, PCB[40] layout and assembly. The PCB was manufactured and assembled in a foundry before delivery. In total 50 boards are produced. The functionality of the platform can be divided into four closely interacting subsystems: computing, sensing, communication, and power supply (see Figure 2.8). We will look into the details of each of these subsystems in the following further details can be found in [57, 59].

Computing an FPGA for hardware development and an MCU with A/D converter for external peripherals. The FPGA operates independently of the ATMega, but is controlled by the ATMega in a number of ways. It features a large number of digital I/O lines, buttons, leds, external UART connection and is connected to an external FLASH. The ATMega powers the FPGA on and off and points the radio interface to either the ATMega of FPGA.

Communication a detachable add on-board with a flexible radio with low level access.

Sensing an add on-board with sensors

Power a power supply allowing battery powered operation while maintaining a steady supply. Further it allows the ATMega to disable the power supply to the FPGA completely.

2.5.2 Design Process

While the subject of sensor platforms has been the subject of numerous recent studies, the design and fabrication process has not. In this section we will focus on some of the lessons learn while building the Hogthrob prototype platform. The design of the prototype platform was carried out as a collaborative effort. The overall design goals were decided jointly by the project partners and implementation was carried out by IO Technologies. Finally the platform was

[40] Printed Circuitry Board

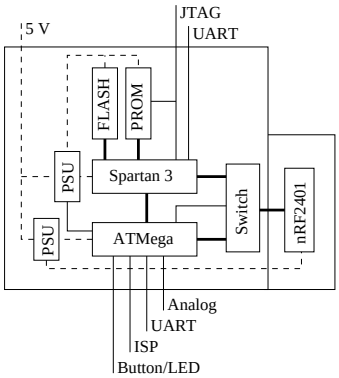


Figure 2.8 *HogthrobV0 components. Dashed lines represent power, full lines represent control.*

tested and bug-fixed by the project partners. We believe this work division is similar to other areas and that the lessons we have learned are valuable to other platform designers.

The design process in interdisciplinary in nature and part of the system cannot be built in isolation. In our case he design was influenced by 3 major parts: i) the application requirements ii) the software design and iii) the hardware constraints.

The prototype platform was designed with a dual purpose: it must serve as a development platform with a fair amount of flexibility and it must be able to carry out simple field test experiments. This choice is not unlike many current generation research platforms, but is unlikely to be valid for large scale deployments.

Quite early in the design process a few of the major components were fixed: the auxiliary MCU (ATMega128) and the FPGA (Xilinx Spartan 3). While the FPGA introduces a unprecedented degree of freedom, the interaction with the external components does not - in particular the ATMega. While the ATMega serves the job as a maintenance MCU for the platform it was chosen primarily simply because of familiarity with this MCU. Cheap, suited and with well proven TinyOS support.

Finally the harsh realities of bringing our wishes together: designing the board layout and connecting the components. This was carried out by I/O Technologies with very little interaction with the project partners. A few PCB's were hand-soldered and tested before the remainder was produced. The final lot was tested and modified appropriately.

The design process lead to a number of unforeseen difficulties, and remedying the mistakes took far larger resources than first planed:

- While the engineers had spent some time attempting to get the inter chip connections right, the platform was plagued with a number of flaws. In particular the MCU/FPGA, and RF interfaces were in nature quite complicated and difficult to visualize without an intricate knowledge of the software that would later control the units.

- In the end the fabrication process delivered bare-boards only. No documentation or development tools accompanied the boards. After the boards were handed over to us we had to rediscover the rationale behind the design, setup a tool chain and write the drivers. Having no prior knowledge or documentation as well as the fair amount of design flaws made this a time consuming process.

- The design testing and fabrication testing was for the most part carried out at a late stage in the design process and without the assistance of the engineers who built the platform.

Sorting features and bugs were up to the software team.

- The design process was much more expensive and time consuming than hoped. A second revision of the board was not carried out for lack of time and money.
- In the end some of the features were designed differently than project partners had imagined.

In general, the stages of the design process were far to isolated and resulted in a sub optimal design and sub optimal use of resources. One of the goals of this platform is to explore the HW/SW boundary, and while this is quite possible this was not a focus of attention during the design process.

2.5.3 Finalizing and Testing HogthrobV0

The board was designed and implemented by an external partner. As such the delivery of the boards was the start of a learning process. The testing procedure is used as a tool to bootstrap this learning process as well as providing a concrete tool for testing the platform. The platform was delivered in two stages. First a few boards were delivered for testing and evaluation. The testing involved developing the software to be run on the platform testing every feature of the platform. In a second stage the 50 boards are produced. The goal of the testing procedure was:

1. acquire the knowledge to be able to utilize the platform and
2. uncover flaws in the design and fix potential problems
3. uncover problems with the individual boards

By designing a general testing procedure that reaches all corners of the design we learn the innards of the platform while producing the tests required to uncover flaws once all 50 boards are delivered. The result of this process is the documentation and the testing procedure code.

The platform was delivered with relatively little documentation. A board schematic, a list of components and the the concept document that formed the order for the engineers. One of the major components in the design that was underestimated at the beginning of the project. The effort required to build learn the intricate details of the platform and assemble a development setup were far grater than envisaged.

Testing

In general the goal of the testing procedure is to ensure that the functionality that we require is working as expected for each board. However it is used in a broader context, to provide examples, to produce documentation, and to discover design blunders. It essential that the procedure is consistent for each board and reproducible such that the result can be verified at a later stage.After the initial stage, the testing procedure is used to verify each board.

When employed on each board the testing procedure should detect design flaws and flaws that stem from manufacturing (such as faulty PCB, or imperfect mounting). In addition it must detect that no mistakes were made during post production modifications. To accomplish these to objectives we adopted a component based strategy, by breaking up the platform as a graph of connected components. For the hardware testing all these components will be hardware component, but a similar technique can be used to for testing software components.

We draw a graph of the available components and implement a test for each of the component interconnects. The graph involves all programmable components such as the MCU and

FPGA, but does not include passive components such as the power sub system or other helper systems. We assume, that all of these subsystems are directly or indirectly controlled by the programmable subsystems.

In this way we test all of the functionalities that involve more than one component. We do not however, test the internals of each component. The internals of each component have been tested by the manufacturer, if it boots and executes code we consider it to be working perfectly. Nor do we test the passive (PSU, etc.) systems in any specific way. If the system boots we assume that the these systems are working perfectly.

Oregano Core

Once the FPGA has been booted it operates independently of the ATMega. It features a large number of digital I/O lines, buttons, leds, external UART connection and is connected to an external FLASH.

The Oregano 8051 IP Core[41] is an 8051 clone in VHDL realeased free of charge under the lGPG licence[42]. In design it is very similar to the 8051 clones that are commercially available today. It has shorter instruction execution time and provides some of the common peripherals: timer, uart. It includes a software boot loader that reads programs from the uart and executes them.

By providing the source code Oregano enables us to modify and adapt the core to our needs. This involves synthesizing it for our Xilinx FPGA and it allows us to simulate the design in a simulator. In either case the design must be setup for the particular tool chain and connected appropriately to the surrounding environment. This includes:

- Mapping the memory areas of the design memory on the board
- Mapping I/O pins to physical I/O pins
- Adjusting and connecting the system clock
- Setting up the project for the Xilinx tool chain (as opposed to the Altera tool chain used by the project).

In our case that involves setting up the project for the simulation and synthesis tool chains, as well as connecting logical signals to the particular incarnations on our FPGA and board.

Xilinx provides a suite of tools for synthesizing designs for Xilinx FPGA's. Most prominent is the Xilinx ISE development and synthesis environment, along with the ChipScope on-chip debug and ModelSim XE digital simulator. This simulator is closely integrated with the Xilix tool chain and we have chosen to simulate Oregano using this tool. While the simulator is a valuable development tool for debugging and testing, but it does not interface with external components or simulation of external stimuli.

2.6 Summary

In this chapter we have reviewed some of the topics related to the Hogthrob project and to this dissertation. We briefly mentioned some of these topics in the introduction and this chapter has discussed some of the vast number of options available to us. The intention is to set the scene for the remainder of the dissertation in a number of specific areas:

[41] http://www.oregano.at
[42] http://www.gnu.org/licenses/lgpl.html

Application Requirements We looked at two classic examples of sensor network monitoring, and compared them to the Hogthrob pilot experiment. In all cases there was no good understanding of the performance of the application prior to the deployment. In the case of Hogthrob and Zebranet it turned our that the performance was good enough, but we argue, that a systematic approach to investigating the performance prior to the experiments would have been helpful in all cases.

System-on-a-chip for Hogthrob We reviewed a set of sensor network motes. This review describes some of the limitations of the currently available hardware and outlines some of the future potential for new platforms. The conclusion for Hogthrob is that none of the generic motes meet the price or performance requirements.

TinyOS We described some of the advantages of TinyOS over other sensor network operating systems. In the following chapters we will take advantage of the unique features of TinyOS in the problems we attack. In particular the unique component model will allow us to introduce a transparent logging layer and the stringent hardware abstraction architecture enables us to port TinyOS to very different platforms in an application agnostic way.

Power estimation We reviewed some of the previous work within power estimation in sensor networks and embedded communities. The review builds the intuition required to describe the power model (Section 2.4.4) that is the implicit foundation for the power estimation technique described in Chapter 3.

HogthrobV0 We outlined the HogthrobV0 platform, that has not been explored fully. The true potential of this platform will be unveiled when combining it with the techniques presented in the following chapters. For example using the vector based methodology to estimate the advantages of certain hardware accelerators implemented on the platform.

Now, that we have set the scene, we will go on to develop our concrete methods. First we will look at performance evaluation in Chapter 3 and we will port TinyOS to a prominent system-on-a-chip platform in Chapter 4.

CHAPTER 3

Characterizing Mote and Application Performance

In this chapter we will present our vector based methodology for sensor mote and application characterization. This method uses a set of benchmarks to describe a sensor network mote and express the hardware utilization of an application in the units of the benchmark. This allows objective comparison of mote hardware and applications, as well as performance prediction of a know application on a new platform.

We will begin by introducing sensor network performance and the related work. Then we will present our methodology and implementation. Finally we will present our experimental results and a comparison of the Sensinode Micro and CC2430 platforms. The ideas and large parts of this chapter has been published as a peer reviewed conference paper[60]. This chapter constitutes an expanded and revised version of the paper.

3.1 Introduction

Sensor networks-based monitoring applications range from simple data gathering, to complex Internet-based information systems. Either way, the physical space is instrumented with sensors extended with storage, computation and communication capabilities, the so-called motes. Motes run the network embedded programs that mainly sleep, and occasionally acquire, communicate, store and process data.

To increase reliability and reduce complexity, research prototypes [34, 75] as well as commercial systems[1] now implement a tiered approach where motes run simple, standard data acquisition programs while complex services are implemented on gateways. These data acquisition programs are either a black box (Arch Rock), or the straightforward composition of building blocks such as sample, compress, store, route (Tenet[34]). This approach increases reliability because the generic programs are carefully engineered, and reused across deployments. This approach reduces complexity because a system integrator does not need to write embedded programs to deploy a sensor network application.

Such programs need to be portable to accommodate different types of motes. First, a program might need to be ported to successive generations of motes. Indeed, hardware designers continuously strive to develop new motes that are cheaper and more power efficient. Second,

[1] See http://www.archrock.com

a program might need to be ported simultaneously to different types of motes, as system integrators need various form factors or performance characteristics.

Handzicki, Polastre et al.[38] address the issue of portability when they designed TinyOS 2.0 Hardware Abstraction Architecture. They defined a general design principle, that introduces three layers:

1. Mote Hardware: a collection of interconnected hardware components (typically MCU, flash, sensors, radio).

2. Mote Drivers: Hardware-specific software that exports a hardware independent abstraction (e.g., TinyOS 2.0 define such Hardware Independent Layer for the typical components of a mote).

3. Cross-Platform Programs: the generic data acquisition programs that organize sampling storage and communication.

We rely on these three layers to reason about mote performance. Whether motes are deployed for a limited period of time in the context of a specific application (e.g., a scientific experiment), or in the context of a permanent infrastructure (e.g., within a building), power consumption is the key performance metric. Motes should support data acquisition programs functionalities within a limited power budget. We focus on the following questions:

1. What mote hardware to pick for a given program? The problem is to explore the design space and choose the most appropriate hardware for a given program without having to actually benchmark the program on all candidate platforms.

2. What is a mote hardware good for? The problem is to characterize the type of program that is well supported by a given mote hardware.

3. Is a driver implemented efficiently on a given hardware? The problem is to conducted a sanity check to control that a program performs as expected on a given hardware.

We are facing these questions in the context of the Hogthrob project, where we design a data acquisition infrastructure. First, because of form factor and cost, we are considering a System-on-a-Chip (SoC) as mote hardware. Specifically, we want to investigate whether Sensinode Nano, a mote based on Chipcon's CC2430 SoC, would be appropriate for our application. More generally, we want to find out what a CC2430 mote is good for, i.e., what type of applications it supports or does not support well. Also, we had to rewrite all drivers to TinyOS 2.0 on CC2430, and we should check that our implementation performs as well as TinyOS 2.0 core. Finally, we would like to use Sensinode Micro as a prototyping platform for our application as its tool-chain is easier and cheaper to use (see Chapter 4 for details). We would like to run our application on the Micro, measure performance, and predict the performance we would get with the Nano.

In this paper, we propose a vector-based methodology to study mote performance. Our hypothesis is that energy consumption on a mote can be expressed as the scalar product of two performance vectors, one that characterize the mote (hardware and drivers), and one that characterize the cross-platform application. Using this methodology, we can compare motes or applications by comparing their performance vectors. We can also predict the performance of an application on a range of platforms using their performance vectors. This method will enable sensor network designers answer the questions posed above. Specifically, our contribution is the following:

1. We adapt the vector-based methodology, initially proposed by Seltzer et al.[92], to study mote performance in general and TinyOS-based motes in particular (Section 3.3).

2. We conduct experiments with two types of motes running TinyOS 2.0: Sensinode Micro and CC2430. We ported TinyOS to these platforms (see Section 3.4).

3. We present the results of our experiments (Section 3.5). First, we test the hypothesis underlying our approach. Second, we compare the performance of the Micro and CC2430 motes using their hardware vectors. Finally, we predict the performance of generic data acquisition programs from the Micro to the CC2430.

3.2 Related Work

The study of sensor networks has been tightly coupled with the study of performance. Most project focus on reducing energy consumption and prolonging battery life. Most project take a relatively simplistic approach and study the impact of a certain subsystem in isolation, for example listing the total energy consumption of a radio using different protocols. Typically, analytical models, simulation or benchmarking are used to study the performance of a program. In addition we employ techniques such as program tracing and workload characterization. In section we will look at some of the related topics to our vector based methodology.

The use of simulation relates a cost model to a simulation of an application run. The precision of the estimate relies on the accuracy of the cost model and the ability to model the environment. One way to distinguish the different approaches is the level of abstraction taken in the cost mode: PowerTossim[94] abstracts very complex operations such as send or compute into a fixed cost based on the state of the peripheral units of the mote. Other projects such as Avrora[100], SensorSim[80], SimpleScalar[4], and others[13, 21, 86, 96, 104] attempt to simulate varying degrees of internal details of the CPU architecture. In our opinion, simulation is best suited for reasoning about the performance and scalability of protocols and algorithms our method is distinguished by being run directly on the mote hardware in question. The framework is easily portable, whereas building a simulation environment for a new platform is time consuming and requires intricate knowledge of the platform. Further simulations are limited to the model of the environment, our methodology allows tracing applications with real as opposed to simulated inputs. The value of real inputs is illustrated by the EmStar[33] framework allowing a hybrid simulation mode mixing real and simulated nodes in a network.

The use of benchmarks to characterize a given hardware platform is a well employed discipline in many areas. Standard benchmarks fall into two categories: application benchmarks (SPEC, TPC), or microbenchmarks (lmbench)[2]. There is no such standard benchmark for sensor networks. Micro benchmarks have been defined for embedded systems, but do not tackle wireless networking or sensing issues: EEMBC, MiBench[36] focus on the automotive and consumer electronics markets, MediaBench[55] focus on multimedia related issues.

In the area of sensor networks no common benchmark has gained high popularity. TinyBench[41] propose a few application and micro benchmarks evaluating code size and energy consumption, the tests are limited to a single mote and does not relate the benchmarks to actual program execution. SenseBench[76] defines workloads as composition of building blocks, and present a set of metrics. The authors focus on the energy consumption of a unit of data (energy per bundle), the code footprint and introduces a metric xRT that emphasizes on event processing rather than execution time. xRT attempts to answer whether or not a given workload can be executed at a given frequency of a mote, rather than the actual speed of a given benchmark.

[2]See http://www.tpc.org, http://www.spec.org, http://www.bitmover.com/lmbench, and http://www.eembc.org/ for details about these benchmarks.

The recent Wisenbench[74] is focused on exploring the impact the CPU or instruction set architecture on application performance. The instruction and memory usage of a wide spread of benchmarks are analyzed, however the use of peripheral units are not covered and the relation to target applications are unclear.

Our work follows-up on the work of Jan Beutel that defined metrics for comparing motes[9]. Instead of using data sheets for comparing mote performance, we propose to conduct application specific benchmarks.

In addition to traditional methods to study hardware and application performance some projects attempt to capture the energy consumption and other parameters of a program running on a target platform. PowerScope[27] instruments an operating system to correlate a Unix-process with a current measurement. Power Meter[48] instruments the power subsystem of a mote to collect a total current consumed during an interval. Within the area of deployment support and test beds motes are often augmented with a second device with a back channel. This second device can act purely as a programming support device or can be augmented with monitoring features. MoteLab[103] adds a serial forwarder, but allows motes to be augmented with a voltmeter. Deployment-support network (DSN)[10] feature a programmable second system that monitors the first, the system is however not featured with power monitoring equipment, it does however allow very fine grained monitoring of the code execution.

Performance estimation is of the essence for real-time embedded systems. The focus there is on timing analysis, not so much on energy consumption. We share a same goal of integrating performance estimation into system design [10].

Our work is a first step towards defining a cost model for applications running on motes. Such cost models are needed in architectures such as Tenet [34] or SwissQM [75] where a gateway decides how much processing motes are responsible for. Defining such a cost model is future work.

3.2.1 Tracing Execution

Program tracing is the subject of numerous studies and techniques. In general, the available tracing methods relies on either *intrusive* or *non intrusive* techniques. Non intrusive techniques includes methods such as in circuit emulators (ICE), hardware counters (e.g. oprofile), etc., while intrusive methods often relies on *code instrumentation* either at compile time or by modifying binary code. Code instrumentation often involves inserting probe points counters (e.g. gprof) this can be automated of manual.

Regardless of the technique each chooses a specific parameter related to executing program: function call, I/O activity, interrupt rate, etc. Furthermore most techniques focus primarily on execution time, while we are interested energy consumption. While the energy consumption with reasonable accuracy can be abstracted as a function of time the use of peripheral units such as radio or adc does not necessarily. Lastly these techniques does not extract the program execution en a platform independent manor that allows one program trace to be transformed from one platform to an other.

3.2.2 Application Specific Benchmarking

The vector-based methodology proposed by Setlzer et al.[92] takes it's starting point in application specific benchmarking. The authors note that traditional benchmarking techniques can be misleading when trying to determine the actual performance based on the performance of a benchmark. Instead the authors argue the case for application specific benchmarking, that is to create a benchmark that resembles the workload of the application. Developing such a

benchmark for each workload is impractical, and the authors propose a methodology that can be applied to any application across domains.

To derive the method the authors note the following observation: in a typical computer system, each different primitive operation, whether at the application, operating system, or hardware level, takes a different a different time to complete. The principle behind this method is to represent the underlying system abstractions as a vector quantity. Each component of this system characterization vector represents the performance of one underlying primitive, and is obtained by running an appropriate microbenchmark. Corresponding to the system vector, a second vector that represent the demand an application places on each underlying primitive. An execution time estimate is derived as a linear combination of the two vectors (the dot product). As an example the authors use this method to analyze the performance of java virtual machine (JVM). By using a suite of microbenchmarks they characterize the performance of the JVM and they use application profiling to quantify an applications use of these primitives. Let $\underline{v}$ be the JVM performance vector and $\underline{a}$ be the application vector. Now, a performance estimate could be derived as the dot product of these two vectors, but in the case of the JVM there is a complication: garbage collection. To model the overhead of garbage collection, let $g()$ model this overhead as a function of the application vector, this overhead is then added as an extra term in the equation:

$$\underline{v} \cdot \underline{a} + g(\underline{a})$$

For certain applications the performance is dependent, not only on the application it self, but on the particular input to the application. To accommodate these types of applications the authors propose characterizing a workload based on a trace. Consider for example a web server—by modeling the workload based on for example a server log, this workload can be replayed and the performance of the system observed. Finally the authors note, that the above methodology assumes that all operations are independent. This is not the case for elaborate system components such as cache, resulting in inaccuracy. To solve this problems the authors propose a hybrid approach in which the system vector also describes cache sizes in the system. This vector and a trace is used as input to a simulator that models the effect of the particular sequence request. The simulator then derives an application vector taking more complicated system behavior into account.

In the following section we will develop this method to suit the needs of sensor network motes and extend the method to include energy. Please note, that the traces we collect in the following are not identical with the trace based benchmark above.

3.3 Vector-Based Methodology

The vector-based methodology[92], consists in expressing overall system performance as the scalar product of two vectors:

1. A system-characterization vector, which we call **mote vector** and denote $\underline{MV}$. Each component of this vector represents the performance of one primitive operation exported by the system, and is obtained by running an appropriate microbenchmark.

   We create two mote vectors, one corresponding to the execution time and one corresponding to the energy consumption attributed to each benchmark.

   $\underline{MV_e}$ (Coulomb) and $\underline{MV_t}$ (Seconds)

2. An application-characterization vector, which we call **application vector** and denote:

$$\underline{AV} \text{ (utilization of system primitives)}$$

Each component of this vector represents the application's utilization of the corresponding system primitives, and is obtained by instrumenting the API to the system primitive operations.

Our hypothesis is that we can define those vectors such that mote performance can be expressed as their scalar product:

$$Energy = \underline{MV_e} \cdot \underline{AV}$$

$$Execution\ time = \underline{MV_t} \cdot \underline{AV}$$

The methodology has several advantages over current methods. First, comparing two motes objectively can be done by comparing their respective mote vectors. By choosing the mote vector components carefully, we illustrate a set of relevant parameters for a mote, instead of looking at a single figure or only CPU performance. Second, by extracting a workload description in the form of an application vector we avoid defining synthetic benchmarks that may or may not give pointers to the performance of a target application, instead we can simply extract a mote independent characteristic of the target application. Third, the methodology allows speculative prediction of performance from mote to mote, note that in the equations above if the mote and application vectors are known we can simply carry out the dot product, this is our estimate.

Our challenge has been to devise a methodology adapted to mote performance. The issues are i) to define the mote vector components, and the microbenchmarks used to populate them, and ii) to create a software framework to record the relevant parameters, and iii) to verify the feasibility of the method by defining a representative application workload, to collect a trace from the instrumented system API, and to convert an application trace into an application vector.

3.3.1 Mote Vector

We consider a system composed of the mote hardware together with the mote drivers. The primitive operations exported by such a system are:

- CPU duty cycling: the network embedded programs that mainly sleep and process events need to turn the CPU on and off.
- Peripheral units: controlled by the CPU through the hardware-independent functions made available at the drivers interface[3].

We choose this system because its interface is platform-independent. This has two positive consequences. First, we can use mote vectors to compare two different motes. Second, the application vector is platform-independent. We can thus use our vector-based methodology to predict the performance of an application across motes.

The mote vector components correspond to the CPU (when active or idle), and the peripheral units (as determined by the driver interfaces). Throughout this Chapter, we use an associative array notation to denote the mote (and application) vector components, e.g., $\underline{MV}[active]$

[3]Note that we assume that the mote hardware relies on a single CPU to control all peripheral units. Peripheral units such as digital sensors might include their own micro-controller. Our assumption simply states that a mote program is run on a single CPU.

corresponds to CPU execution, $\underline{MV}[idle]$ corresponds to CPU sleep, $\underline{MV}[PU_i]$, correspond to peripheral units primitives where PU_i is for example ADC sample, flash read, flash write, flash erase, radio transmit, radio receive.

We need to define a metric for the vector components. The two candidates are energy and time. We actually need both: (a) energy to compute the scalar product with the application vector and thus obtain mote performance, and (b) time to derive the platform-independent characteristics of an application (see Section 3.3.2). We thus need to define a microbenchmark for each mote vector component for which we measure time elapsed and energy spent. We distinguish between the energy mote vector, noted $\underline{MV_e}$, and the time mote vector, noted $\underline{MV_t}$.

The microbenchmarks must capture the performance of the system's primitive operations. The first problem is to represent CPU performance. The most formidable task for the CPU in a sensor network application is to sleep. This is why we distinguish sleep mode from executing mode in the mote vector. For the applications we consider, a single sleep mode is sufficient. Defining a microbenchmark to define the energy spent in sleep mode is trivial. However, we wish to use the time mote vector to compare the time spent in sleep mode by different motes. Intuitively, the time spent in sleep mode is a complement of the time spent processing. As an approximation, we thus consider that $\underline{MV_t}[idle]$ is the complement of $\underline{MV_t}[active]$ with respect to an arbitrary time period (fixed for all mote vectors), and that $\underline{MV_e}[CPUsleep]$ corresponds to the energy spent in sleep mode during that time.

The second problem is to define an appropriate representation of CPU performance (in executing mode). Unlike peripheral units, for which drivers define a narrow-interface, the CPU has a rich instruction set. It is non-trivial to estimate the CPU resources used by a given application as it depends on the source code and on the way the compiler leverages the CPU instruction set. We choose a simple approach where we use a microbenchmark as a yardstick for the compute-intensive tasks of an application. We thus represent CPU performance using a single vector component. There is an obvious pitfall with this approach: we assume that the distribution of instructions used by the microbenchmark is representative of the instructions used by the application. This is unlikely to be the case. We use this simple approach, despite its limitation, as a baseline for our methodology because we do not expect CPU utilization to have a major impact on energy consumption. Our experiments constitute a first test of this assumption. Obviously much more tests are needed, and devising a more precise estimation of CPU utilization is future work. It remains to be shown what impact the imprecise CPU estimation has on the total estimate in a realistic scenario, given that sensor mote applications often are non computing intensive.

The third problem related to the microbenchmarks is that driver interfaces often provide a wide range of parameters that affect their duration and energy consumption. Instead of attempting to model the complete range of parameters, we define microbenchmarks that fix a single set of parameters for each peripheral unit primitive. Each peripheral unit microbenchmark thus corresponds to calling a system primitive with a fixed set of parameters, e.g., a microbenchmark for radio transmit will send a packet of fixed length, and a microbenchmark for ADC sampling will sample once at a fixed resolution. We believe that this models the behavior of sensor network application that typically use a fixed radio packet length or a particular ADC resolution. This method can trivially be expanded to cover a finite set of configurations by defining a vector component per parameter set (e.g., replacing radio transmit with two components radio transmit at packet *length_1* and radio transmit at packet *length_2*).

For the sake of illustration, let us consider a simplistic mote with a subset of the TinyOS 2.0 drivers, that only exports two primitives: ADC sample and radio transmit (tx). The associated

time mote vectors will be of the form:

$$\underline{MV_t} = \begin{bmatrix} t_{active} \\ t_{idle} \\ t_{adc} \\ t_{tx} \end{bmatrix}$$

Where the mote vector components correspond to the time spent by the mote running the CPU microbenchmark, to the time spent in sleep mode (the complement of the time spent running the CPU benchmark with respect to an arbitrary time period that we set to 20 s), to the time spent running the ADC benchmark, and to the time spent running the transmit benchmark.

In order to express mote performance as the scalar product of the energy mote vector and the application vector, we need the components of the mote vectors to be independent. This is an issue here, because CPU is involved whenever peripheral units are activated. Our solution is to factor CPU usage in each peripheral unit component. As a consequence, the mote vector component corresponding to CPU performance ($active$) must be obtained without interference from the peripheral units. Another consequence is that we need to separate the CPU utilization associated to peripheral units from the pure computation, when deriving the platform-independent characteristics of an application. We thus register CPU time when benchmarking each peripheral unit primitive. We denote them as $CPU[PU_i]$ for each peripheral unit primitive PU_i.

We detail in the next Section, how we use those measurements when deriving the application vector from a trace.

3.3.2 Application Vector

Our goal is to characterize how an application utilizes the primitives provided by the underlying system in a mote independent way. In essence the application vector is a charaterization of a specific workload. In the context of sensor networks, workload characterization is complicated (i) because motes interact with the physical world and (ii) because the network load on a mote depends on its placement with respect to the gateway, and (iii) because different motes play different roles in the sensor network (e.g., in a multi-hop network a mote located near the gateway deals with more network traffic than a mote located at the periphery of the network). In the following we extract the mote vector from trace of the application utilization. This is a different approach than a traditional benchmark comparing how two artificial programs compare on different platforms—by extracting a concrete workload based on an application the benchmark is one step closer to running the actual application and measuring the performance.

The application vector denoted $\underline{AV}$ is of the same form as the mote vector, however each component describes how a specific application utilizes a particular peripheral unit using the benchmarks from the mote vector as metric:

$$\underline{AV} = \begin{bmatrix} active \\ idle \\ adc \\ tx \end{bmatrix}$$

In order to collect the trace we consider that a sensor network application can be divided into representative epochs that are repeated throughout the application lifetime. For example, the application we consider in the Hogthrob project consists of one data acquisition epoch[4],

[4] A sensor network deployed for collaborative event detection will typically consist of two epochs: one where motes are sampling a sensor and looking for a given pattern in the local signal, and one where motes are communicating once a potential event has been detected.

where an accelerometer is sampled at 4 Hz, the samples are compressed, stored on flash when a page is full, and transmitted to the gateway when the flash is half-full. While sampling is deterministic, such an epoch is non-deterministic as compressing, storing or transmitting depends on the data being collected, and on the transmission conditions. Obviously, tracing an application throughout several similar epochs will allow us to use statistics to characterize these non-deterministic variations.

The question is now, how do we derive an application vector from a running program?

3.3.3 Application Trace

For each epoch, we trace how the application uses the CPU and the peripheral units. We consider the state of the mote to be described by the state of the CPU and the state of the peripheral units, expressed as a linear combination of the mote vector components, and it is this state that the trace records. More precisely the trace records the total time spent by the mote in each possible mote state, defined by the combination of active mote vector components ($active$ that represents the compute-intensive operations, $idle$ that represents the CPU in sleep mode, and PUi that represents a peripheral unit interface call). $\underline{T}$ is of dimension 2^m, where m is the dimension of the mote vector. Some of the mote states will not be populated because they are mutually exclusive (e.g., $active$ and $idle$), or because the driver interfaces prevent a given combination of active peripheral units. By recording this trace we capture any parallel activity between peripheral units—we will take advantage of this feature in a moment when discussing peripherals competing for a shared resource.

Let us get back to the simple example we introduced in the previous section. The trace vector for an epoch will be of the form:

$$\underline{T} = \begin{bmatrix} active \\ idle \\ adc \\ tx \\ adc \;\&\; tx \\ active \;\&\; adc \\ active \;\&\; tx \\ active \;\&\; adc \;\&\; tx \\ \cdots \end{bmatrix}$$

Now the problem is to transform, for each epoch, the trace vector into a platform-independent application vector. The application vector $\underline{AV}$ has same dimension m as the mote vector, and each application vector component corresponds to the utilization of the system resource as modeled in the mote vector. The application vector components have no unit, they correspond to the ratio between the total time a system primitive is used in an epoch, by the time spent by this system primitive in the appropriate microbenchmark (as recorded in the time mote vector $\underline{MV_t}$). Note that if the driver primitive is deterministic, then the ratio between the total time spent calling this primitive in an epoch and the microbenchmarking time is equal to the number of times this primitive has been called. However, drivers typically introduce non-determinism,because the scheduler is involved or because drivers embed control loops with side effects (e.g., radio transmission control that results in retransmissions).

As stated in the introduction we assume a high level approach to recording the trace: a peripheral unit is considered active when the corresponding driver primitive is activated and is considered to be deactivated when this driver primitive signals this to the application. This high level approach introduces a complication when considering peripheral units executing in

parallel. Consider a state indicating that two peripheral units are active. Essentially this means that both driver interfaces have been activated in the same time interval. If the underlying hardware unit are truly parallel, they will execute accordingly. However, if the two units are not able to execute in parallel this merely indicates that the driver interface have scheduled each to run in isolation within this period of time. As we are mapping the components of the trace to independent units, we must decide how to attribute a recorded time to two different units. We encode this information in the architecture matrix **AM** for each mote, essentially taking the architecture specific details out of the trace. This approach is key, as the competing resources are likely to be different from mote to mote.

We use a linear transformation to map the trace vector onto the application vector. This transformation can be described in three steps:

1. We use an **architecture matrix** of dimension m,2^m to map the trace into a vector of dimension m, the **raw total time vector**, where each component correspond to the total utilization of the CPU and peripheral units. The architecture matrix encodes the definition of each state by defining a mapping between the recorded time spent in specific states and the mote vector components that are mutually independent.

   Note that this combination depends on the architecture of the mote. For example, a SPI bus might be shared by the radio and the flash. In this case, the time spent in a state corresponding to radio transmission and flash write is spent either transmitting packets or writing on the flash (there is no overlap between these operations). We assume fair resource arbitration and consider that both components get half the time recorded in the trace. In case of overlap between operations, both get the total time recorded in the trace.

   In our simplistic example, assuming that a resource is shared between the radio and the ADC (a bus for example), the architecture matrix will be of the form:

$$\mathbf{AM} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & \frac{1}{2} & 1 & 0 & \frac{1}{2} \\ 0 & 0 & 0 & 1 & \frac{1}{2} & 0 & 1 & \frac{1}{2} \end{bmatrix} \begin{matrix} \\ \\ \\ \\ \vdots \end{matrix}$$

2. We use a **CPU matrix** $CPU[k]$ to factor out of the $active$ component the time spent by the CPU controlling the peripheral units. The CPU matrix, of dimension $m \times m$, is diagonal except for the row corresponding to the $active$ component. This row is defined as 1 on the diagonal, 0 for the $idle$ component, and $-\underline{CPU}[k]/\underline{MV}[k]$ for all other components. When multiplying the total time vector with the CPU matrix, we obtain a **total time** vector where the $active$ component corresponds solely to the compute-intensive portion of the application:

$$\underline{TT} = \mathbf{CPU} \times (\mathbf{AM} \times \underline{T})$$

   Using again our running example, we have a CPU matrix of the form:

$$\mathbf{CPU} = \begin{bmatrix} 1 & 0 & -\frac{CPU[adc]}{\underline{MV_t}[adc]} & -\frac{CPU[tx]}{\underline{MV_t}[tx]} \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3. We use the time mote vector to derive the application vector. The basic idea is to express the application utilization of the system primitive as the ratio between total time per component, and the time spent running a benchmark. We define the inverse mote vector, $\underline{MV^{-1}}$, as a vector of dimension m where each component is the inverse of the time mote vector component (this inverse is always defined as the time mote vector components are always non zero). We define the application vector as the Hadamard product of total time vector with the inverse mote vector.

   With our running example, we obtain the equation:

$$\begin{bmatrix} totalactive/\underline{MV_t}[active] \\ totalidle/\underline{MV_t}[idle] \\ totaladc/\underline{MV_t}[adc] \\ totaltx/\underline{MV_t}[tx] \end{bmatrix} = \begin{bmatrix} totalactive \\ totalidle \\ totaladc \\ totaltx \end{bmatrix} \circ \begin{bmatrix} 1/\underline{MV_t}[active] \\ 1/\underline{MV_t}[idle] \\ 1/\underline{MV_t}[adc] \\ 1/\underline{MV_t}[tx] \end{bmatrix}$$

More generally, we derive the application vector from the trace vector using the following linear transformation:

$$\underline{AV} = (\mathbf{CPU} \times (\mathbf{AM} \times \underline{T})) \circ \underline{MV^{-1}}$$

And we obtain the mote performance as the scalar product of the application vector with the energy mote vector:

$$E = \underline{AV} \cdot \underline{MV_e}$$

3.3.4 Example

Let us consider an illustrative example. We picture a simple mote, that can sample and send. We imagine a set of mote vectors and an imaginary trace, using the method described above we derive the mote independent application vector and estimate performance on a second mote. Note that we leave out the states that are mutually exclusive (and therefor empty), for the sake of brevity. In the following we will list the equations along with the matrices that we construct for the sake of this example.

We imagine recording a trace $\underline{T}$ of some program on this simple mote. This mote is constructed such that send and sample compete for a shared resource equally (say a bus), this is expressed in the architecture matrix **AM**. Using this matrix we can derive a raw time $\underline{R}$ assigning time to each peripheral unit. In the equation below the peripheral units are represented using one bit and the CPU using two, the state number is simply the concatenation of these bits. The matrices leave out the empty rows and columns for brevity.

$$\mathbf{AM} \times \underline{T} = \underline{R}$$

Bit no.	**Comp**
1	Active
2	Idle
3	Sample
4	Receive

Architecture Matrix
$\mathbf{AM} =$

$$\begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 0 & 0 & 1 & 1 & \frac{1}{2} & \frac{1}{2} \end{bmatrix}$$

State		**Trace** $\underline{T} =$
1	(1000)	11
2	(0100)	8
5	(1010)	2
6	(0110)	0
9	(1001)	4
10	(0101)	0
13	(1011)	4
14	(0111)	0

Raw Time $\underline{R} =$

$$\begin{bmatrix} 21 \\ 8 \\ 4 \\ 6 \end{bmatrix}$$

We further imagine that this mote uses 1 unit of CPU time for both send and sample, this expressed in the CPU matrix **CPU**. Using this assumption we now correct the raw time measurements to form the total time vector $\underline{TT}$

$$\mathbf{CPU} \times \underline{R} = \underline{TT}$$

$$\mathbf{CPU} = \begin{bmatrix} 1 & 0 & -\frac{1}{2} & -\frac{1}{2} \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \underline{R} = \begin{bmatrix} 21 \\ 8 \\ 4 \\ 6 \end{bmatrix} = \underline{TT} = \begin{bmatrix} 16 \\ 8 \\ 4 \\ 6 \end{bmatrix}$$

With the adjusted total time, we can now derive the mote independent application vector $\underline{AV}$. To do this we need the mote vector $\underline{MV}$. We imagine that this mote uses 8 ms to execute the active benchmark, leaving 2 ms for idle in a 10 ms epoch, and 2 ms to execute send and sample benchmarks expressed.

$$\underline{TT} \circ \underline{MV^{-1}} = \underline{AV}$$

$$\underline{TT} = \begin{bmatrix} 16 \\ 8 \\ 4 \\ 6 \end{bmatrix} \circ \left(\underline{MV_t} = \begin{bmatrix} 8 \\ 2 \\ 2 \\ 2 \end{bmatrix} \right)^{-1} = \underline{AV} = \begin{bmatrix} 2 \\ 4 \\ 2 \\ 3 \end{bmatrix}$$

Now imagine a second mote with a mote vector $\underline{MV_t^{'}}$. Using this mote vector and the mote independent $\underline{AV}$ we can now estimate the execution time $\underline{TT^{'}}$ of the program on a second mote.

$$\underline{AV} \circ \underline{MV_t^{'}} = \underline{TT^{'}}$$

$$\underline{AV} = \begin{bmatrix} 2 \\ 4 \\ 2 \\ 3 \end{bmatrix} \circ \underline{MV_t^{'}} = \begin{bmatrix} 7 \\ 3 \\ 3 \\ 3 \end{bmatrix} = \underline{TT^{'}} = \begin{bmatrix} 14 \\ 12 \\ 6 \\ 9 \end{bmatrix}$$

This simple examples illustrates the simple, yet powerful principle of the method. It models the usage of each peripheral unit and captures contention for shared resources in a simple fashion. The question is course will these assumptions be sufficient in practice? And further what are the peripheral unit components required to capture the performance of a mote reliably?

3.4 Implementation in TinyOS 2.0

In the previous Section we have described our methodology in general terms; in this Section we will formulate the method into a concrete implementation. We base this implementation on a set of example applications, that we consider to be representative to a class of sensor network

applications: data acquisition. We use these application as a basis for defining the mote vectors that we will use in our experiments.

To populate these vectors we create a software framework that will allow us to collect that are related to either the mote (mote vector, cpu vector, architecture matrix) or the application (trace, application vector). Collecting the mote vectors is done once using a set of carefully constructed benchmarks; one for each entry in the vector, while the software framework for collecting traces is constructed so that it can record any application without modification. The two sets of measurements are very similar, and is composed of the same primitives, they do however differ substantially in the time span of the measurement. Consider for example collecting the duration of a packet transition. The actual primitive operation lasts only perhaps 1 ms, while an application sending and receiving data could run for minutes, days or longer. This results in a slightly different setup for the two measurements: for the mote vectors we use a high level of manual interaction, while the program traces are recorded in a highly automated fashion.

3.4.1 Applications and Mote Vectors

We use simple data acquisition applications as workload for our experiments. We build them from building blocks: sample, compress, store, and send. We create 4 applications that increase the parallel behavior of these tasks from isolation to parallel sample and transmission:

SampleCompressStore is a simple state machine, that runs each step in isolation. As each sample is retrieved, it is then compressed, and once 10 samples are retrieved they are stored to flash. This cycle is repeated 9 times.

DataAcquisition extends the state machine from SampleCompressStore to retrieve the data from flash and transmit it. Again, each step in isolation.

SampleStoreForward is similar to DataAcquisition, except without the compression step.

DataAcquisitionAdv performs the same tasks as DataAcquisition, but interleaves the sample and transmit processes. Store is done in isolation.

For our first experiments, we want a deterministic workload that exhibits reproducible results. One important source of variance in a sensor network applications is the environment. We choose a simple network topology and transmission scheme. Data is transmitted in 384 byte chunks corresponding to 3 in-air 802.15.4 packets. The transmission does not expect acknowledgment that a packet is received, but only wait for the channel to be cleared (CCA) before sending. Sampling is at 10 Hz and for compression we use the Lz77 algorithm.

Mote Vectors and Benchmarks

The vector component are chosen by analyzing the components used by the applications. As a result, we choose the following components for their mote vectors: $active$, $idle$, adc, $radio\_receive$, $radio\_transmit$, $flash\_read$, $flash\_write$, and $flash\_erase$. Doing so, we leave some of the peripheral unit primitives out of the mote vector (e.g., the primitives to set or get the channel on the 802.15.4 radio) and unused peripherals. The time spent executing primitives left out are factored as CPU execution time, while the unused peripherals are only considered to contribute the idle power consumption. We also leave timers, UART and general IO pins out of the mote vector. The time spent in the timers is factored in the CPU idle component. We leave general IO pins out because we do not use LEDs, or digital sensors. Similarly, we do not use the UART. Note that we do not consider a specific sensor connected to the ADC.

The benchmarks we defined for these mote vector components are:

- A compression algorithm to characterize CPU execution. This component contains a mix of integer arithmetic with many loads and stores and some function calls. Using this algorithm is a baseline approach.
- Simple function calls with a fixed parameter for each peripheral unit primitive[5]. Note that benchmarks, in particular for the radio and flash, contain some buffer manipulation. These are measured as $CPU[PU_i]$ (see Section 3.3.1).

$$\underline{MV_t} = \begin{bmatrix} active \\ idle \\ adcsample \\ radioreceive \\ radiotransmit \\ flashread \\ flashwrite \\ flasherase \end{bmatrix}$$

Benchmarks

In order to populate the mote vectors we create a set of benchmarks. Each benchmark exercises a particular feature of the platform. During the benchmark we record the duration of the events using the framework below and the energy consumption is measured externally. Recording the change in energy consumption assumes that total energy consumption is a linear combination of each of the sub components. Such that for example running the ADC and radio can be expressed as the sum of the active CPU, ADC contribution and radio contribution.

The peripheral units often provide a wide range of parameters that affect the duration and energy consumption of the platform, say radio packet length or ADC precision. Instead of attempting to model this dependence we capture the duration for a single set of parameters, this models the behavior seen in many sensor network application: using fixed radio packet length or a particular ADC resolution. This method can trivially be expanded for a fixed number of parameter (say two packet lengths, etc.).

TestCompression This application reads data from the UART, compresses it and sends it back. Only the compression part of the application is timed during the benchmark. It has a node side (TestCompression) and PC side (node-comm). The PC side reads data from a text file, transmits it to the mote in chunks of 256 bytes. The mote stores it in a buffer, compressed it and sends it back. This application was developed for the Hogthrob pilot experiment[71] and ported to TinyOS 2 and the motes that will be described in the following section.

TestSeqAdc, TestSeqTx, TestSeqFlash These applications activate the respective peripheral unit one: sample, send, read flash, write flash, erase flash. Each uses a fixed set of parameters (e.g. one 16 bit sample, one 128 byte packet, one 256 byte flash page).

In order to factor out the CPU component that is attributed to each of the operation we record the utilization of the CPU during the benchmarks. These measurements make up the CPU vector.

[5]The source code is available through the project website http://www.tinyos8051wg.net

3.4.2 Capturing the Trace

Now with the components of the mote vector defined we need to record the trace. Recall, that we consider the trace to record the total time the mote spends in each possible state. Each state is the combination of the mote vector components, corresponding to peripheral units. We represent each unit using one or more bits and the state is simply the concatenation of each of the bits. In order to attribute the time spent in each state it is sufficient to consider only the transition between states and accumulate the time between transitions to the appropriate state.

To identify the state transitions we use the following observation: for the systems we consider all peripherals are controlled by the CPU. The state transitions will thus occur at the point in time when the CPU changes the state of a peripheral unit. This transition is executed by a line of code and we use this line a *probe* or *measurement point*. The trace must record the duration between the probe points and accumulate it to a specific state.

For example an analog to digital converter could be started by some internal register being written and ends when it signals that a value is ready. For this example the measurement point would be the line writing to the register and in the interrupt handler serving the event. For units that are either on or off we assign one bit, for units that are able to operate in more than one state we assign one bit for each state (e.g. radio receive, radio transmit). Some peripheral units operate in discrete events (e.g. sample ADC), while others are variable (e.g. time spent in idle)—regardless of the mode of operation we record the time in the same manner, allowing us to treat all units the same.

One could imagine inserting counters at these probe points and reading report the counters at program termination. This imposes an overhead in terms of timer administration and counter administration. Whether this overhead is significant of course depends on the speed of the MCU and the duration of the events. In our case we are considering short events on an MCU with a low clock rate, potentially leading us to a situation where there simply isn't any instructions left to manage the counters. As we shall see in a moment the operations we are considering are in the micro-second range (the shortest are between 1 ms and a few hundred μs). Having a clock rate off for example 1 MHz is not uncommon for a mote, the clock period would then be 1 μs. This means the even if updating counter at the beginning and end of an event can be implemented efficiently we only have in the order of a few hundred to a thousand clock cycles available during the event for timer manipulation and to carry out the regular load[6].

Instead we output each bit of the state as an IO pin, using a second mote, which we call *LogRecorder*, that records the state transitions. Updating a pin only takes a few instructions I/O pins are available on virtually any MCU used for mote design. On the other hand this solution is more cumbersome as it requires a second device for timekeeping. This mechanism is very similar to the monitoring techniques devised for deployment-support networks[10]. Implementing an alternative on chip timekeeping layer for ease of use would be relatively straightforward.

In order to identify and collect these measurement points we rely on a mix of manual and automatic software instrumentation of TinyOS 2.

3.4.3 TinyOS API Instrumentation

Drivers in TinyOS comes in the form of components that provide interfaces to a set of operations related to a specific hardware device. We chose this point of entry as it allows to insert a platform independent layer between the actual component implementing the driver and the

[6]Updating a 16 bit counter array with and elapsed time $t_1 - t_0$ about 15 instructions on the MSP430. Toggling a pin requires one.

```
interface Read<val\_t> {
  command error\_t read();
  event void readDone( error_t result, val_t val );
}
```

(a) Read interface

```
generic configuration ReadLoggerC(typedef width_t) {
  provides interface Read<width_t> as ReadOut;
  uses interface Read<width_t> as ReadIn;
}
implementation {
  components new ReadLoggerP(width_t);
  ReadOut = ReadLoggerP;
  ReadIn = ReadLoggerP;

  components new PinC() as Pin;
  ReadLoggerP.GeneralIO -> Pin;
}
}
```

(b) ReadLogger configuration

Figure 3.1 *a) TinyOS 2 Read interface b) The logger layer to be sandwiched in between a component providing the Read interface and a component using the Read interface. Notice that this component has a ReadIn and a ReadOut interface corresponding to the interfaces of the components that it is inserted between. It uses a pin as the logging mechanism.*

component using the driver. The CPU and the peripheral units are instrumented slightly differently:

- For the peripheral units, we introduce a platform-independent layer between the component that provides the driver interface and the component that uses it. As an example consider reading a value from the ADC using the TinyOS 2.0 *Read* interface (see Figure 3.1). This interface starts an ADC conversion with a *Read* command and returns with a *readDone*, We insert a layer that records the time elapsed between the *Read* command is called and the *readDone* event is received. This is obviously an approximation of the time during which the ADC is actually turned on. In a similar fashion we instrument the FLASH and radio interfaces.

- The MCU is modeled slightly different that the peripheral units. We wish to capture the time spent in sleep mode and in executing mode. As mentioned we consider only a single sleep mode, while many MCU provide a rich set of sleep modes, in practice the combination of peripheral units often limit the choice to a single sleep mode.

  TinyOS has a simple task scheduler that puts the MCU into sleep mode when the task queue is empty. The microprocessor is awoken via interrupts generated from internal or external peripherals. We consider the time from entering sleep mode to the time the interrupt handler is executes as $idle$ and time the interrupt to the sleep mode as $active$.

  We instrument the TinyOS scheduler and each interrupt handler to record the timing of these events. This is done using a simple script that automatically inserts modifies the

scheduler and modifies the interrupt handler.

In order to collect this trace, we encode each state as a combination of bits (our mote vector is of dimension 8) we thus use 8 bits to encode the states.

3.5 Experimental Results

We applied our vector-based methodology to two motes: Sensinode Micro, a Telos-like mote, and CC2430, which is the basis for a new generation of commercial motes[7]. We ported TinyOS 2.0 on both platforms. In the following chapter we will describe our port of TinyOS to the 8051 platform, while the Sensinode Micro is similar enough to the popular Telos mote, that most code can be shared.

Before describing our results and the implementation framework we will present a little background on the two motes.

3.5.1 CC2430 and Sensinode Micro

As a SoC Texas Instruments's CC2430 has a small form factor (7x7 mm) and promises to be mass-produced at a lower price than complex boards. Motes built around the CC2430 might constitute an important step towards reducing the price of sensor networks. The CC2430 is composed of the 8051 MCU with a wide range of common on-chip peripherals as well as an 802.15.4 radio very similar to the Texas Instruments CC2420. We run the system at 32 MHz. The CC2430 differs from the platforms on which TinyOS has been implemented so far in two important ways: the system architecture and the interconnect to the radio.

The Intel 8051 MCU architecture was designed in the early eighties and many oddities from this era remain. Not only is it an 8 bit, CISC style processor with a Harvard architecture[8], but the main memory is further subdivided into separate address spaces that differ in size, are addressed differently and vary in access time. Simply put, the 8051 defines a fast memory area limited to 256 bytes, and a slow memory area of 64 KiB. In addition to variables, the fast access area contains the program stack. This limits the program stack to less than 256 bytes depending on the amount of variables in this area. Commonly, activation records of functions are placed on the stack, thus potentially limiting the call depth critically. To circumvent this problem, some compilers (e.g. Keil) place stack frames in the slow data area, which imposes a high cost for storing and retrieving arguments that do not fit in registers when calling a function. The slow access RAM also penalizes dynamic memory allocation, and context switches and thus favor an event-based OS with static memory allocation such as TinyOS.

Because CC2430 is a SoC, there is no bus between the MCU and the radio. The MCU controls the radio via special function registers (instead of relying on a SPI bus as it is the case on Telos and Micro motes for example). The other peripheral units (ADC, UART, timers, flash, and pins) are accessed in the 8051 MCU as in other micro-controllers such as the MSP or ATmega.

The Sensinode Micro is built around the 16 bit, RISC style MSP430 MCU with combined code and memory spaces (Von Neuman). The platform can run up to 8 MHz, but we choose 1 MHz in our experiments. Apart from the built in common peripherals of the MSP, it features the Texas Instruments CC2420 radio which is connected though an SPI bus.

[7] We experimented with a CC2430 development kit. Using commercial systems based on CC2430, such as Sensinode Nano, is future work.

[8] Code and data are located in separate memory space

3.5.2 TinyOS 2.0 on CC2430 and Micro

TinyOS 2 has been designed to facilitate the portability of applications across platforms. First, it is built using the concept of components that use and provide interfaces. TinyOS is written in nesC, an extension of C that supports components and their composition. Second, TinyOS implements the Hardware Abstraction Architecture[38]. For each hardware resource, a driver is organized in three layers: the Hardware Presentation Layer (HPL) that directly exposes the functions of the hardware component as simple function calls, the Hardware Abstraction Layer (HAL) that abstracts the raw hardware interface into a higher-level but still platform dependent abstraction, and the Hardware Independent Layer (HIL) that exports a narrow, platform-independent interface. The TinyOS 2.0 core working group has defined HIL for the hardware resources of typical motes: radio, flash, timer, ADC, general IO pins, and UART.

Porting TinyOS 2.0 on CC2430 consisted in implementing these drivers[9]. For the timers, pins, UART and ADC we used the TinyOS HIL interfaces *Alarm/Counter*, *Read*, *GeneralIO* and *SerialByteComm* respectively. However, in two cases we choose to diverge from the common interfaces to build interfaces that better suits our needs (see Chapter 4 for details).

Note that we did not need to change the system components from TinyOS 2.0. However, supporting a sleep mode on the CC2430 requires implementing a low-frequency timer. On the pre-release CC2430 chips we used for our experiments, timers do not work properly. This is work in progress, as a consequence our experiments are conducted without low-power mode on the CC2430.

The main challenges we faced implementing TinyOS 2.0 drivers on CC2430 were to (i) understand the TEP documents that describe the core interfaces as we were the first to port TinyOS 2.0 on a platform that was not part of the core, and (ii) to define an appropriate tool chain. Indeed, the code produced by the nesC pre-compiler is specific to gcc, which does not support 8051. We had to (a) choose another C compiler (Keil), and (b) introduce a C-to-C transformation step to map the C file that nesC outputs into a C file that Keil accepts as input (e.g., Keil does not support inlining, the definition of interrupt handlers is different in Keil and gcc, Keil introduces compiler hints that are specific to the 8051 memory model). TinyOS for 8051 is detailed in Chapter 4 and the approach to porting TinyOS is detailed here [58].

Because the Micro has many similarities with the Telos mote, on which TinyOS 2.0 was originally developed, porting porting TinyOS 2.0 was a simple exercise. However, the wiring of the radio does not feature all of the signals available on the Telos mote, meaning that the radio stack could not be reused. We implemented the simple MAC layer, *SimpleMac*, and simple flash layer *SimpleFlash* described above.

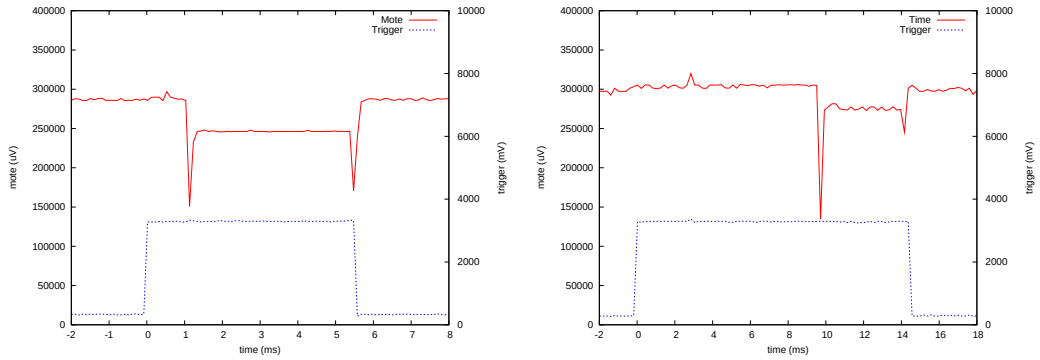
3.5.3 Experimental Setup

To collect traces and measure energy consumption we create the test setup depicted in Figure 3.3. The setup measures the power consumption of the supply line to the mote. This has the advantage that all components are included in the measurements such that the results mimic what we would be able to measure in the field - this includes the power supply unit (PSU) which in it self often has a non negligible power consumption. On the other hand this setup shields some of the transitions that we would expect to see as transitions on the power-line, because the power subsystem distorts or delays the actual current consumption of the internal components.

We use the PC-oscilloscope PicoScope 3204[10] which provides a sufficient resolution in terms

[9] For details, see http://www.tinyos8051wg.net

[10] http://www.picoscope.com



(a) CC2430 (b) Micro

Figure 3.2 *Recording the energy consumption, during one 128 byte packet transmission for the CC2430 and Micro.4 respectively. The right hand axis shows the voltage over a 10 ohm resistor in series with the mote, the left hand axis shows the voltage of the trigger pin. The energy is measured on the low side of the circuit. The radio starts out in receive mode and transitions to send mode, which uses less energy than receive on these motes. The oscilloscope uses the buffer to show the energy before and after the trigger event (t=0).*

of sample rate and buffer size at a low cost. The oscilloscope can either sample continuously offloading data via USB or sample at a high rate to a buffer and offload later. In this case the oscilloscope fills a buffer of 256 k, 8-bit samples with up to 50 MHz sample rate (20 ns sample interval) when a trigger event occurs.

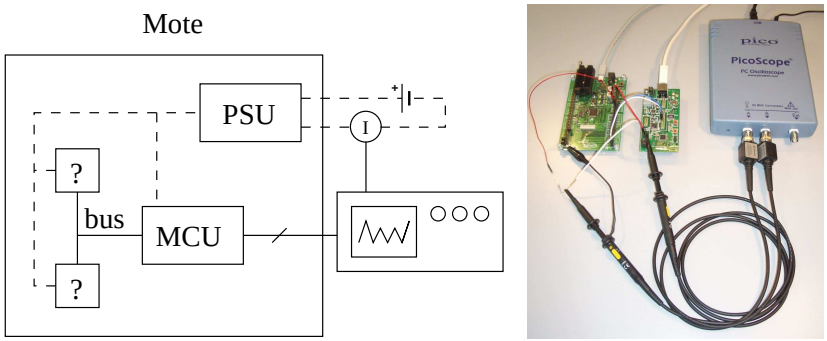
LogRecorder

We use a second mote to record the state transitions that are output from the platform under investigation, we call this mote the *LogRecorder*. We connect the 6 IO pins as well as a trigger pin that is connected to a timer-capture pin. The instrumented platform output a state change by changing a particular bit and by flipping the state of the trigger pin. This causes a timer-capture event in the LogRecorder recording the time of the event with high accuracy.

The Micro.4 provides a 32.768 kHz clock crystal that provides high-accuracy timer ticks, we further calibrate the timer ticks using the oscilloscope (the observed error is below 0.1 ‰). With a 16 bit built in timer the maximum duration of a state is 2 s.

While the TinyOS timer support is able to extend the 16 bit timer values to 32 bit, this does include the capture features. We extend the TinyOS support to return 32 bit extended timer values. This gives us a maximum event length of app. 36 hours.

The vector based approach deals with aggregated execution times, however it does not deal with the frequency and duration of events. As we shall discuss in a moment, this approach may conceal important aspects of program execution. In particular when we speculatively transfer a mote vector from one mote to an other this aggregate value may be insufficient in some cases. To investigate this claim we augment the LogRecorder with an additional feature: along with the regular trace we record a profile of the event lengths for this particular application. We create a set of buckets corresponding to time intervals and count the number of events that correspond to each interval. In addition to shedding light on program behavior this feature will aid a program developer understand in larger detail how a program executes.



(a) Schematic (b) Actual setup

Figure 3.3 *a) Depiction of the instruments used to record the trace, power consumption is measured externally prior to the power consumption unit (PSU), and the trace is collected from the MCU and correlated with the energy b) The actual test setup showing the PicoScope 3204, a CC2430 development kit and a Micro.4 mote as LogRerecorder*

3.5.4 CC2430 and Micro

We ran the benchmarks described in the previous section on both the Micro and CC2430 motes. The time and energy mote vectors we obtain are shown in Figure 3.5 as spider charts. The results are somewhat surprising. CC2430 is much faster than the Micro when running the benchmarks and transmitting packets. Slow memory accesses is compensated by the high clock rate and direct access to the radio speeds up packet transmission. It means that the CC2430 can complete its tasks quickly, and thus be aggressively duty cycled. In terms of energy, we observe that:

1. CPU operations are two to three orders of magnitude more expensive on the CC2430 than on the Micro. This is due to the high clock rate (which guarantees fast execution) and to the overhead introduced by the slow access RAM.

2. Flash operations show an uneven pattern. The CC2430 is faster for all operations, but for write and delete it is 3 orders of magnitude faster, while it is only about 3 times faster while reading. This results in the CC2430 being considerably cheaper for write and delete, but actually more expansive for read.

   At first this result lead us to check our implementation (which is a positive results in itself). We did not find any errors, but the result could be influenced by the fact that write and delete are implemented using hand coded assembly on the CC2430, but not on the Micro.4.

   On the other hand it could be a simple difference in hardware architectures, letting the MSP430 favor reads by 3 orders of magnitude as opposed to write could simply be a matter of choice.

3.5.5 Performance Prediction

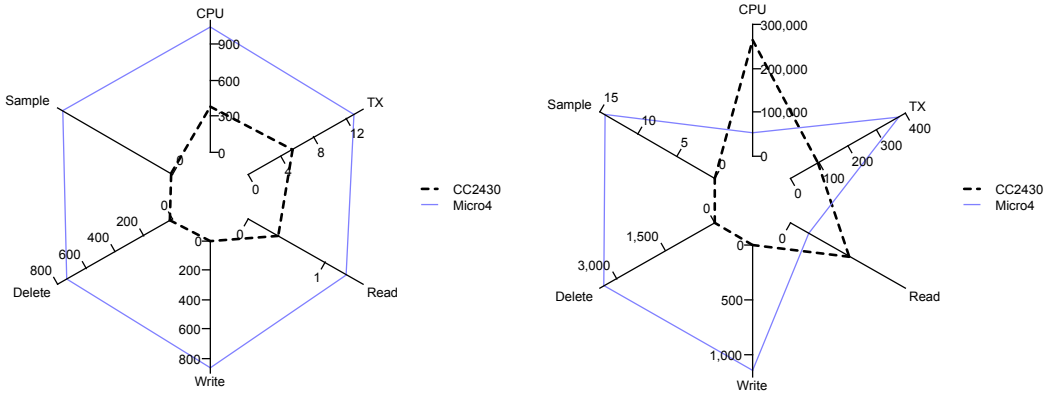
We used our methodology to derive the application vectors for the four data acquisition applications described in the previous Section. The results are shown in Figure 3.7.

The profiles we get for the applications correspond to what we expect. Indeed, the application vector components for the ADC, flash and radio operations correspond roughly to the

$$\begin{bmatrix} \\ MCU \\ Idle \\ Rx \\ Tx \\ Sample \\ Flashread \\ Flashwrite \\ Flasherase \end{bmatrix} \quad \begin{matrix} \text{CC2430} \\ MV_t = \\ \hline \\ 38.054 \\ 961.45 \\ 10 \\ 5.4927 \\ 0.22886 \\ 0.39670 \\ 1.4190 \\ 18.486 \end{matrix} \quad \begin{matrix} \text{Micro.4} \\ MV_t = \\ \hline \\ 104.15 \\ 845.86 \\ 10 \\ 12.874 \\ 19.099 \\ 1.2755 \\ 868.65 \\ 735.50 \end{matrix} \quad \begin{matrix} \text{CC2430} \\ MV_e = \\ \hline \\ 264.93 \\ 6697.1 \\ 219.00 \\ 94.475 \\ 0.073923 \\ 0.10195 \\ 0.2363 \\ 7.8011 \end{matrix} \quad \begin{matrix} \text{Micro.4} \\ MV_e = \\ \hline \\ 53.843 \\ 278.61 \\ 298.42 \\ 376.91 \\ 14.305 \\ 0.031888 \\ 1144.01 \\ 3369.89 \end{matrix}$$

(a) Ordering (b) Time (ms) (c) Energy (mC)

Figure 3.4 *Mote vectors, listed with 5 significant figures. Idle is not implemented on the CC2430 resulting in the same power consumption as active mode. Note that receive has been normalized to 10 units and that the cpu-benchmark is scaled by a factor of 1000 as the benchmark is considerably longer than the following example applications.*

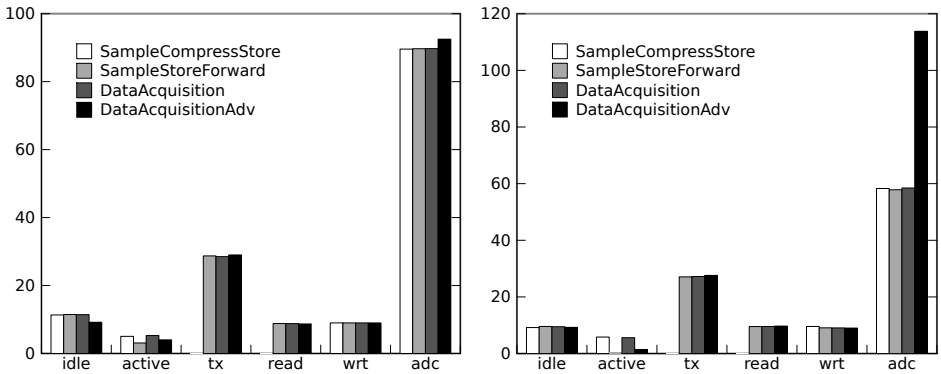


(a) Time Mote Vectors (ms) (b) Energy Mote Vectors (mC)

Figure 3.5 *Time and energy mote vectors for CC2430 and Micro. Note that receive has been left out as none of our applications use receive, and idle has been left out as idle is not implemented on the CC2430.*

	DataAcquisitionAdv		SampleCompressStore	
Interval (ms)	Micro.4	CC2430	Micro.4	CC2430
0-19	0	214	0	160
20-39	95	13	199	16
40-59	36	11	0	14
60-79	154	0	154	10
80-99	0	0	0	0
100-300	32	0	36	0
total	*317*	*238*	*389*	*200*

Figure 3.6 *Event frequencies for two applications. The motes perform the same applications, but differ in the interface to the underlying hardware. Not surprisingly the slower Micro.4 mote shows a shift to slower events, but the results also shows that the difference in architecture also results in a significantly higher total number of events for the Micro.4.*



(a) Micro (b) CC2430

Figure 3.7 *Application vectors for CC2430 and Micro. Note that receive has been left out as none of the application receive data. The unit on the y-axis is the mote vector primitives, such that app. 30 tx corresponds to 30 executions of the tx benchmark sending 384 byte chunks.*

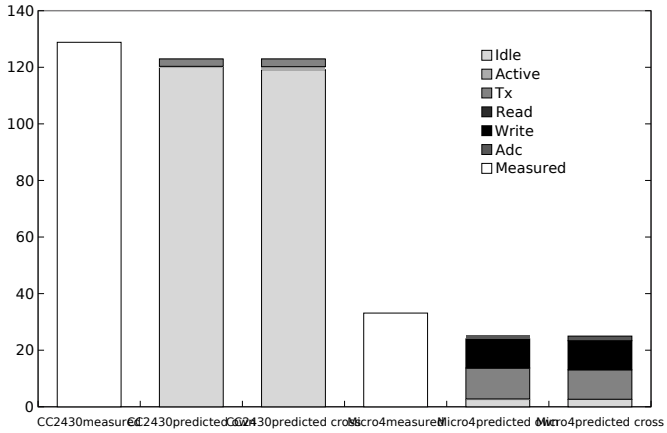


Figure 3.8 *Energy measurements and estimates*

number of samples, flash and radio operations issued by the applications. The application vector is designed to be platform-independent. We thus expect that the application vectors derived from the CC2430 and Micro are similar. The good news is that they are at the exception of the ADC component. This is either a measurement error, a software bug in the driver, or a hardware bug. We focused on this issue and observed that the time it takes to obtain a sample on CC2430 varies depending on the application. Two different programs collecting the same data through the same ADC driver experience different sampling times. We observed as much as 50% difference between two programs. We believe that this is another hardware approximation on the CC2430.

Our initial hypothesis is that the energy spent by an application on a mote can be estimated using the scalar product of the application vector with the mote vector. We computed the energy estimate for the *DataAcquisitionAdv* application and we compared them to the measurements we conducted directly on the motes (using an oscilloscope). The results are shown in Figure 3.8.

The estimations are well into an order of magnitude from the actual energy consumption. This is rather positive. As expected, the contribution from the CPU in active mode is insignificant. The poor performance of the CC2430 is due to the fact that we did not implement sleep mode support on the CC2430. Much more work is needed to test our methodology. This experiment, however, shows that we can use our method to prototype a data acquisition application with the Micro and predict how much energy the CC2430 would have used in the same conditions.

3.6 Limitations

In this chapter we have described our approach to sensor mote benchmarking, and implemented a setup that explores some of the potential of the approach. We have not, however sufficiently explored the approach to be able to make any conclusions to the accuracy and limitations in the general case. With the above experiments, we point to the following:

Accuracy In this work we have not explored the accuracy of the method. We chosen a set of

examples, that illustrates that the method is feasible, but this does not answer what the accuracy of the method is. We have relied on a number of assumptions all attributing to the total error of the vectors and estimates. Exploring and bouning the sources of errors is essential to use this method in a broader perspective.

Architecture matrix We modeled the use of shared resources of a sensor mote. This assumes that the use of a shared resource within a given interval can be divided by a simple fraction. This is a much simpler approach to modeling nontrivial system behavior than the simulation based approach originally proposed[92]. The question is of course if this approach is sufficient, and what impact this has on the overall accuracy of the methodology, in a practical setting.

Shared resources We have chosen a simplistic solution to modeling shared resources. This approach captures and translate the performance penalty when transferring application vectors between motes. The question is, is this simple method sufficient? And how do we determine the coefficients of the architecture matrix such that the results are reliable and accurate?

CPU scaling In the approach we split the CPU activity in two: i) the computational time, ii) the CPU time attributed to driver primitives. This split avoids counting the same attribution twice, but it also which avoids a caveat: the drivers that are required on one platform may be completely different then those required on an other and there is no way to derive performance of a driver on one platform from the performance of a driver on an other platform—the two are simply different programs with no relation.

For the computational part we assumed a linear scaling of cpu consumption from one mote to an other. This is a very crude mapping, in our example we have used it to map from a 16 bit RISC architecture to an 8 bit CISC architecture—it is highly unlikely that a linear mapping exists that accurately captures the architectural differences. Whether this approach sufficient approach requires further investigation.

For the applications we looked at the energy consumption attributed by the CPU was infinitesimal in the overall energy budget, making the accuracy of the CPU approximation less important to the accuracy of the overall system performance estimate.

Linear energy consumption We have assumed that the composition of energy consumption for each peripheral unit is linear. Whether this is sufficient requires more investigation.

Consider for example the fact that the power supply subsystem efficiency is non linear. Essentially the amount of waste energy dissipated in power conversion varies with the load, meaning that the power consumption attributed to one component may depend on which other components are active.

Infeasible workload The method does not provide a clear indication of whether a given workload is feasible for a mote. In the ideal case if we collect a trace on a mote from an epoch and estimate same epoch (active + idle) to be longer than desired on a target mote the workload is clearly not feasible. However, there may be cases where this is not the case.

In our examples there are a number of overheads left out. A mote may simply be unable to support certain workload. For example a workload may require an event rate that is simply not possible using a particular architecture.

In our approach we have augmented the benchmark with a profile of the event length and frequency of an application. The profile will aid a developer in understanding the

performance of a given application and provide further hints to whether a given workload overloads a particular mote.

3.7 Conclusion

In this chapter we have presented a new method for characterizing mote and application performance using a set of vectors. Our approach is based on the hypothesis that mote energy consumption and execution time can be expressed as the scalar product of two vectors: one that characterize the performance of the core mote primitives, and one that characterizes the way an application utilizes these primitives.

We have argued that the method is superior to current sensor mote benchmarking techniques. It is based on the principle of application specific benchmarking, giving a higher level of insight to how a particular application will perform rather than a generic workload. One important feature is to model how a particular application utilizes the subsystems of a mote—this is a different approach than looking at the performance of each subsystem in isolation.

We have shown that the method can support a prototyping approach, by estimating the performance on a target mote based on the performance of a prototype.

By basing the method on application traces, this method can expand the time frame of the collected measurements, and be used to collect traces of applications *in situ*.

This method is distinguished from previous methods in a number of ways:

- This benchmark allows application specific benchmarking for sensor network applications. A benchmark is only useful if it conveys information indicating how a system will handle a particular workload. To solve this problem this methodology uses an abstract representation of a workload as a basis (an application vector).

  This allows workload descriptions to be based on actual *in situ* experiments as opposed to describing synthetic workload that may or may not relate to the actual workload. Further this increases the possible time frame of the benchmarks.

  To characterize an a application we capture a trace of the state of the mote during program execution, this trace is then used to derive the workload description.

  The state of the mote is model by the use of peripheral units—the state of the mote is considered to be described by the combined state of the CPU and the peripheral units.

- This benchmark uses a vector, rather than a single number to describe mote hardware (a mote vector). The vector is composed of a set of micro benchmarks profiling the significant subsystems of a sensor mote: the peripheral units.

  This comparison is based on our benchmarking approach above and allows a more objective and accurate comparison of mote features. This is distinct from earlier approaches[9] by relying on measurements rather than datasheets and by attributing time and energy to system primitives rather than merely listing features.

This approach covers an entire system, including MCU radio and other peripheral as opposed to investigating or benchmarking each in isolation.

- The method supports a prototyping approach. By extracting an abstract workload description we are able to speculatively estimate the expected run time and energy consumption on a given mote.

  Combining the mote vector and application vector yield an estimate of energy consumption and execution time. By capturing the energy usage of the mote during benchmark execution, we use the benchmark as a cost model that allows us to map the mote independent application vector to any mote, that has been benchmarked using this approach.

  Estimating the performance on a mote requires that a mote vector is built. This can be done by running the suit of benchmarks, or it can be based on a estimate that is yet to be built.

Much more experimental work is needed to establish the limits of our approach. Future work includes the instrumentation of an application deployed in the field in the context of the Hogthrob project, and the development of a cost model that a gateway can use to decide on how much processing should be pushed to a mote.

3.7.1 Contributions

In summary, our contributions concerning performance are:

A new methodology for sensor mote benchmarking We proposed a new method for sensor mote benchmarking, based on earlier work by Seltzer et al.[92]. We adapted this method for sensor networks and extended it with the notion of energy—the most prominent metric for motes.

We enhanced the method with the use of an architecture matrix, that models the behavior of shared resources. This is much simpler and more elegant model than the original simulation based approach.

An experimental verification of our methodology We verified our methodology by implementing an experimental framework, this verification consisted in implementing a framework for instrumenting TinyOS programs and collecting measurements on two sensor motes.

The implementation is publicly available through the contribution section of TinyOS[11] and consists of two parts: i) an intermediate logging layer for tracing application behavior, that is inserted between drivers and applications in TinyOS ii) the LogRecorder a separate mote that collects traces based on the output from the logging layer.

A quantitative comparison of two sensor network motes We compare two motes that we consider representative for existing motes (the Sensinode Micro.4) and emerging motes (the CC2430).

3.7.2 Future Work

In this chapter we have outlined our method and shown that it is feasible, however as we have pointed out there are some limitations to the method and some limitations to the choices we have made in this chapter. Using this method in a broader perspective exploring these

[11] http://www.tinyos.net

limitations are essential. This being said we believe that the full potential of this method has not been explored, either in sensor networks or within other areas.

- Much more experimental work is needed, in particular learning the possibilities and limitations of our approach.

  In particular the use of a simple CPU yardstick was clearly a first approach. First it is unclear whether this crude approach is sufficient, exploring the limits to this method and possibly a more representative yardstick is future work.

- In this work we focused on a single operating system (TinyOS) and a few motes. Increasing the parameter space is a next step in uncovering the potential of this methodology. This includes comparing motes and mote operating systems.

  An interesting project would consist in collecting samples from a larger selection of real motes in a similar fashion to the Sensor Network Museum[12] Having a database of mote vectors for popular platforms would give application implementers valuable insights when selecting or building platforms.

- The mote vector selection is key in our approach. In this chapter we analyzed a few applications that we believe are representative for a class of sensor network applications. However, they are not representative of all applications. Investigating in general setting how to select appropriate vectors

- We have not investigated the benefits of being able to capture traces of applications *in situ* tracing as opposed to tracing in the lab. Using real input as opposed to simulated inputs could potentially lower some of the uncertainties of modeling the environment, and certainly lower the workload involved in understanding and modeling the inputs.

- The use of vector based benchmarking, could be useful to a much broader audience. In our case we have used it to model the use of peripheral units, but we are confident that this methodology can be applied to a number of unrelated areas.

  For example consider the following example. To evaluate the performance of a database system, one might describe a set of benchmark queries that are expected to be part in a real workload. Now for a new database system one could execute this suit of benchmarks and do back-of-the-envelope calculations that would yield insights to how a real workload would perform. Or instead an approach similar to the one presented here could be explored—expressing the benchmark queries as components of a vector.

  Similarly we believe, that the method we propose by correlating program execution to certain probe points is applicable in other domains for performance or debugging purposes.

[12] http://www.btnode.ethz.ch/Projects/SensorNetworkMuseum

CHAPTER 4

TinyOS for 8051

The use of system on a chip devices that has been envisioned for some time in the sensor network community, but the concrete implementations are few. In this chapter we pick a point in the sensor network mote design space and test how the sensor network operating system TinyOS cope with this point in the design space.

We present our implementation of TinyOS on the 8051 based Texas Instruments CC2430. We will introduce the 8051 (MCS51) architecture and present the CC2430 implementation. We will go on to present TinyOS and the recent advances in TinyOS 2. Lastly we will discuss the difficulties in adapting TinyOS to the CC2430 and 8051 in general.

4.1 Introduction

The vision that spawned sensor networks as a research field nearly a decade ago was the dream of *smart dust*[51]. Smart dust consisted of cheap devices sprayed over an area, some devices would be lost, but the sheer number would be enough to solve complex tasks. It seems clear that the current state of the art of sensor networks, fall short of the vision of ubiquitous, plentiful smart dust. One way to move towards this direction is the use of system on a chip devices (SoC), however few project have ventured down this path and projects who have taken on this challenge have not gained popularity.

In this chapter we take a pragmatic approach: we pick a commercially available system on a chip and explore the challenges in implementing a senor network operating system. We choose the Texas Instruments CC2430 that features a radio similar to the highly popular CC2420 radio and we choose the popular sensor network operating system TinyOS. The questions for us are:

- How will TinyOS cope with this architecture? Previous attempts at bringing TinyOS to similar platforms have failed, can we remedy the faults of these projects and build a reliable TinyOS implementation?

- What is the limitations of the architecture? Specifically: what software components can we run on this platform?

One of the popular sensor network operating systems in the recent years has been TinyOS. TinyOS is a component based operating system, that provides a level of encapsulation that should facilitate a level of abstraction hiding the details of the underlying architecture.

- The required abstraction for building platform independent applications using TinyOS have not been available until recently.

  While the component language of TinyOS (NescC) provides the tools to write platform independent abstractions, this has not been practically possible. The level of abstraction must consistent and implemented uniformly across platforms, in order for applications to move freely across platforms.

  TinyOS 1 did not to a large extent focus on communicating common fixed interfaces among platforms - there was some consensus between a few select platforms, but in general this was not the case. TinyOS 2 introduced the TinyOS Enhancement Proposals (TEP) in part to resolve this issue. By providing a platform independent abstract description of core TinyOS features platforms would agree. Maturing these documents and bringing existing platforms in line with the text is a long process which is only just seeing the fruits of the effort.

  It is essential, that we our implementation take advantage of this development.

- The current implementations have not made it beyond the prototype stage. In our opinion the authors did not focus on providing a code base or community that would encourage new users to move to this platform.

  Some of the features of TinyOS that have ensured the popularity of TinyOS is the documentation and the access to lively support forums. We do not believe that the authors attempted mimic this model persuading users to invest time in the projects. While the code the ports is still available on the web through open source licenses, free of charge—it seem clear that they have been abandoned.

- At the time of writing the MSC51 based CC2430 is not supported by TinyOS. Supporting any new platform involves providing the appropriate programming abstractions as well as providing a tool chain. The TinyOS tool chain is centered around the GCC family of tools, which does not support the MCS51 architecture.

  Providing a reliable tool chain and code base and tool chain for the 8051 is of the essence.

Our goal is to create a TinyOS 2 port that does not exhibit these flaws. It must adhere to current TEP's, be well documented and provide some of the support tools that has made TinyOS popular. In this way we hope to leverage the popularity and familiarity of TinyOS and develop confidence in the reliability of our work. This goal provide a set of different challenges than merely a proof of concept implementation.

4.2 Related

In recent years the research community has spawned a large number of platforms within the field of sensor networks. Even so only a few platforms and micro controller architectures have gained widespread popularity. The absence of system on a chip devices, is particularly striking. System on a chip (SoC) is one of the most promising directions to meet the price, size and energy requirements of sensor network applications. Recent advances in chip technology has made SoC devices commercially available, but still they are struggling to become popular within sensor network research.

In particular the combination of CPU and radio on a single chip is interesting for sensor networks. This combination can potentially reduce price and energy consumption, further integrating the two more tightly can potentially lead to more efficient behavior. Designing SoC devices specifically for sensor networks and specific applications is a topic that shows great potential, but for now we will take a pragmatic approach and look at commercially available devices. This of course limits the choice of processors and radio to the general choices made by the manufacturers. The design of system on a chip devices is a research topic on its own and is beyond the scope of this work.

The processor choices for sensor networks have been limited to a few architectures: the Texas Instruments MSP and ATMega AVR by far have been the dominating architectures, however at the time of writing no SoC devices featuring any of the two are available.

On the other hand SoC devices based on the MCS51 architecture are emerging at this time. Several groups are looking at the Nordic Semiconductor nRF24e1[78] and Texas Instruments CC2430[16] in relation to sensor networks. Ideally picking an architecture is a matter of choice, but it is our belief that more often than not other circumstances narrow the available choices. Processor choices in sensor network in general is discussed here [69].

The key problem as we see it is the software support system allowing application implementers to move easily to these new emerging platforms. TinyOS was designed with the AVR and MSP architectures in mind, and moving beyond these has proved challenging. Previous attempt to port TinyOS to the PIC[54, 67, 68] and MCS51[77, 81, 82] architectures has failed to become popular outside of their initial publications.

The question is of course: why have all the promising projects failed in becoming popular and widespread? This is of course an impossible question to answer, but we can speculate that it has not been the goal of these project to build a foundation for further projects. On the other had we will strive to build this project in a transparent and open way hopefully attracting interested parties.

In parallel with the work we present here Beck et. al[6] has explored a similar path. The workshop paper describe a very similar approach. It is however unclear what the focus of the implementation is: will it span other 8051 implementations, does it follow current TinyOS standards, will the code be made available. On the contrary we will attempt to mimic some of the trait that has made TinyOS itself popular: the high level of confidence, documentation and support.

4.3 Portability and Sensor Networks

A strategy common to most projects is to rely on prototype or proof of concept implementations. Meaning that the implementations are a first step, but that some parameters are not desirable. We believe that the success of sensor networks is dependent on sensor networks advancing from this prototype state to a mature technology.

In the Hogthrob project we have argued the case for a highly configurable prototype platform, how does this lay the foundation for future sensor networks? We postulate that the answer lies in portability. In this chapter we will present a highly portable programming environment that will allows easily to move an implementation from prototype to final implementation. It will allows us transition to new platforms, it will allow us explore new areas that were previously out of reach.

Portability in S/N:

1. Explosion in mote hardware

2. Prototype vs. real implementation

3. Variety of operating systems

Recent advances in chip technology has enabled small, cheap, power efficient integration of multiply sensor network system components on a single die. These system on a chip (SoC) components will present opportunities for sensor network motes to move to the next level in terms of miniaturization and price.

Our motivation stems from the Hogthrob project, seeking to mount a miniature sensor ear tag on a Sow. This task involves minimizing size, cost and maximizing lifetime. Recently commercial system on a chip (SoC) have become commercially available, however these systems have yet to be widely used within the TinyOS community. As sensor networks evolve in this direction we believe it is essential that TinyOS evolves towards these platforms.

A popular processor in this domain is the MCS51 architecture, which is unsupported by TinyOS at this point. We believe that TinyOS is well suited for this platform and will provide an elegant framework that in may ways will be superior to existing frameworks for this platform. Supporting this hypothesis is a number of proof of concept implementations for this architecture by us and others, none of these has gained wide spread use. We believe these attempts failed to gain acceptance because of the following deficiencies:

4.3.1 Portable Embedded Software

While the dream of representing programs in an abstract, platform independent manner lives, in practice it is rarely the case. In practice most programs are dependent on one or more of their underlying assumptions. The key question is what level of abstraction is appropriate? For embedded applications the added overhead of even a slim abstraction may be undesirable. A key element within the framework is the ability to move identical applications from platform to platform.

In general creating portable applications is a challenging task. In the desktop world the number of parameters is usually kept to a bare minimum (tool chain, OS, arch.). Quite often the parameters are kept to a bare minimum (win32 vs. POSIX, x86). In the embedded world, the playing field is considerably more open. With a emphasis on cheap tailored solutions, the diversity in all aspects of system design.

1. CPU architecture

2. operating system

3. tool chain

Our solution is to focus only on TinyOS. The recent TinyOS is designed with portability in mind and features a very low overhead. By relying on TinyOS we also take advantage of the body of work that is built around TinyOS, allowing these applications to seamlessly migrate to our new platform.

4.4 The 8051 Architecture

The term 8051 is usually used to refer to a family of architectures evolved from the Intel MCS51 architecture (including for example the Intel 8052 and similar architectures from other manufacturers. The Intel MCS51 architecture evolved from earlier Intel micro controller architectures

Mote	Micro	Mica2	Eco	Nano
MCU	MSP430F1611 *)	ATM128l *)	nRF24e1	CC2430
Radio	CC2420	CC1000	SoC	SoC
Clock speed (Mhz)			-20	16, 32
RAM / Prg. mem. (KiB)	10 / 48	4/ 128	4 / ext. (up to 4)	8 / 32, 64, 128
Active/Sleep (mW/μW)	0.5 mW / 2 uW	60 mW / 75 uW	/ 2 uW	/ 0.9 uW (w. RTC)
Wakeup Time (uS)			6	180
Mem banking	no	no	no	code $>$ 128 KiB
RF data rate	250 kbps	19.2 kbps	1 Mbps	250 kbps
Address space	16 bit (van Neumann)	16 bit (Harvard)	16 bit (Harvard)	16 bit (Harvard) +code banking
CPU Registers	16, 16 bit	32, 8 bit	3 + 4x8, 8 bit	3 + 4x8, 8 bit
No. IO ports	48	53	11	21
Ext. interrupt sources	16 (2 ints)	8 (8 ints)	2 (2 ints)	21 (3 ints)
ADC resolution	12 bit (8 inputs)	10 bit (8 inputs)	10 bit (9 inputs)	8-14 bit (8 inputs)
Timers channels	2x 16 bit	2x8 bit, 2x 16 bit	1x8 bit, 2x 16 bit	2x8 bit, 1x16 bit + MAC timer
Toolchain	gcc	gcc	Mangle	Mangle
Chip price (USD)	14 (MSP430) + 7 a) (CC2420)	15 (ATM128l) + 7 a) (CC1000)	5 b) + 1 (ROM) a)	9.40 a) (F64)
Mote price (USD)	TelosB: 93 $ c)	Mica2: 155 c)		Nano: 75 d)

Table 4.1 *Comparison of popular sensor network MCUs. The table compares key parameter, however some details are left out for brevity, for example the ATMega128 extends the code memory space by using word addressing for code memory. All prices are single chip prices Dec. 6. 2007, a) mouser.com b) nuhorizons.com c) xbow.com d) sensinode.com *) Numbers from David Culler: Wireless Sensor Networks - the next IT revolution (35th Korea Electronics Show).*

in the early 1980ies, and to this day continues to be highly popular. The original chip has gone trough 25 years of chip evolution and the clones today share little but the instruction set in common with their ancestors. Current clones are tailored to a multitude of applications and vary substantially in the code execution time, peripherals, price and more.

The MCS51 architecture was designed as an 8 bit, CISC style processor with variable opcode length, instruction execution time, etc. It uses an Intel style accumulator register with an additional 4 banks of 8 working registers. The memory architecture is based on the Harvard principle separating code and data memory spaces. The code and data memory is limited to 16 bit (64 KiB) each. The code size limit is often circumvented by using *code banking* in which portions of the memory space is exchanged, with this technique the limit of the code memory is expanded to 128 KiB.

The 8051 architecture diverges substantially compared to the more modern RISC based architectures that have been popular within sensor networks (such as ATMega AVR and Texas Instruments MSP). The 8051 remains interesting because of the SoC combinations that are emerging. In the low price, low performance domain the 8051 remains interesting despite the aging architecture. Table 4.1 compares two 8051 SoC variants to the two commonly seen AVR and MSP variants. Note, that in addition to reduced price, combining components further reduces component count saving space and further costs.

In the following we will discuss some of the significant differences in the 8051 architecture.

4.4.1 Memory Model

The memory model of the MCS51 architecture contains a combination of several techniques, that can only be presumed to be the result of simple solutions to age old problems. Apart from splitting code and data memory, the MCS51 architecture further defines a set of memory spaces and addressing modes. Each of the memory spaces differ in size are addressed differently and vary in access time. The fast access portion of memory *data* (or scratchpad) is limited to 256 bytes, making it likely that many applications will have to place larger items in the slow access *xdata* (originally *extended* memory). This means the architecture imposes an overhead on larger data items as opposed to smaller.

In addition to variables the small data memory area this area contains the program stack. This limits the program stack to less than 256 bytes depending on the amount of variables in this area. Commonly activation records of functions are placed on the stack, if this were the case this could limit the call depth critically. One solution is to a stack frame for each function in the xdata area, which imposes a high cost for storing and retrieving arguments that does not fit in registers when calling a function. This solution is chosen by the Keil compiler, imposing an overhead on function calls and complicates re-entrant functions.

To further complicate matters many 8051 implementations uses the technique of banked memory to extend the code memory space without extending the architecture. Essentially an extra set of addresses bit are used in addition to those presented by the architecture. In this was certain parts of the main memory can be exchanged at run time. This causes further headaches for C programmers, in the following we will disregard the use of banked memory, possibly limiting the amount of memory available to programs[1].

In general the division of memory spaces complicates the use of C-pointers. In case of the Harvard architecture the practical solution chosen by many compilers is to simply let a pointer refer to the data portion of memory and transparently handle the code memory. In case of the 8051 architecture this is not sufficient: the data portion of the memory is further subdivided and a pointer now has to distinguish between these areas. The solution chosen by most compilers is to extend the C language with memory type specifiers and new primitive types. For example the MCU control registers (special function registers) are located in distinct *sfr* memory space, accessing this area is commonly done by declaring a special *sfr* variable.

Each compiler uses a different way to annotate access to this and other areas, but in general a special C-extension is available. For example in the Keil syntax, the *sfr* variable type an memory type specifier is used to access this are. For example declaring the address of a register P0 (not assigning to it) looks as follows.

```
sfr  P0 = 0x80;
```

Other variables can be declared to belong to a specific area using a memory type specifier, for example to place an array in the xdata are looks like the following (still in the Keil syntax):

```
int xdata i[255];
```

By extending C with extra annotations it allows a C programmer to access these portions of memory, but it breaks the pointer notation of C. Consider a *void* pointer, to which address space does it refer? The answer is compiler specific and is often tuned at compile time, by selecting a default area. Some compilers add the notion of a *universal* or *generic* pointer that pads the address with memory type information. While making all pointer assignments safe, this structure, adds an overhead (both in terms of run time and space).

[1] Some compiler support exist for automatically handling code banking, but using the code banks incur an overhead, and still require some manual annotation.

Lets go on to review some of the specifics of the CC2430 variant that further complicates matters.

4.4.2 8051 Compilers

As a popular device in the embedded arena, a large number of compilers exist for the 8051 architecture. Unfortunately the highly popular GCC compiler is not one of them, for various technical reasons. This has implications for TinyOS as we shall see in a moment, but for now we will introduce 3 popular compilers here that we will focus on later:

Keil PK51 A commercial compiler and graphical integrated development environment for Windows ($\mu Vision$). The tool kit is split in a graphical user interface and a command line based compiler that operates independently of the graphical user interface. Pricing information is not public, but our single user edition was priced at app. 3000 EUR and an additional 500 EUR annual fee.

IAR C/C++ compiler for 8051 A commercial integrated development environment (IDE) for Windows. Projects are created using the graphical user interface, but an existing project can be combined using a command line tool. Pricing is not public, our single user edition was priced at app 1000 EUR, an evaluation version is available free of charge.

SDCC is open source, command line based compiler suit. It is free of charge using the Gnu GPL license, for most Unix like platforms (Linux, MacOS C, Solaris, etc.) as well as Windows.

In relation to TinyOS the important feature for all of these compilers is the ability to operate independent of the graphical user interface, making scripting simpler.

Choose: memory model large!

4.4.3 CC2430

The Texas Instruments CC2430 is a system on a chip (SoC), with a CPU and a radio. It has few external components and is built with miniaturization in mind. The CC2430 commonly seen features such as timers, bus controllers as well as internal FLASH, analog to digital converter (ADC) and a radio very similar to the popular TI CC2420 radio. It is built around the Intel MCS51 (8051) architecture, while most peripherals have little or nothing in common with the original architecture. It does, however resemble other 8051 enabled devices from TI, such as the CC1010, CC2431, CC2510, etc[2].

The CC2430 extends the 8051 memory model in two ways: all the memory spaces are mapped into the xdata memory space (see Figure 4.1) and it provides the optional *unified* mapping of the code memory space. These mappings eases programming as well as providing a few advantages. The first allows the DMA controller (that operates in the xdata) area to access all parts of memory including flash. The unified mapping allows code execution from all parts of memory. In addition the CC2430 adds a transparent instruction cache primarily as a power optimization.

The built in radio is programmed in a fashion very similar to the CC2420. Using a similar set of op-codes (or *command strobes*) the operation modes, settings, etc. of the radio is changed. The op-codes are transmitted from the CPU to the radio using a memory mapped register, where the CC2420 uses an external SPI bus. This register can be setup to transfer the content of packet using DMA transfer, thus offloading the CPU. The op-code language of the CC2430 is so similar

[2]See http://www.texasinstruments.com for further

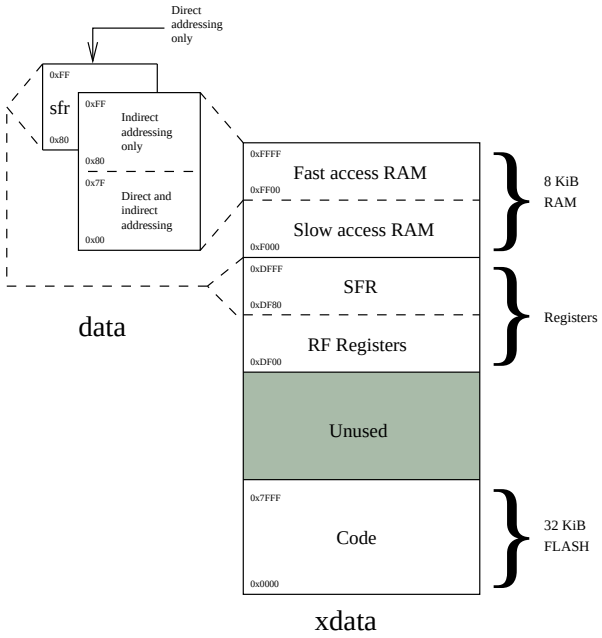


Figure 4.1 *CC2430 memory mapping (32 KiB flash version), in our examples we will use only the lower portion of code memory, despite the fact that the CC2430*

that of the CC2420 radio, that a CC2420 program can be used directly with little or no changes; only changing the interface. In addition the CC2430 contains a simple command co-processor (*command strobe processor*) that can operate independently of the CPU. This simple structure enables relatively complex operations without waking up the CPU, for example rejecting a packet destined for a different recipient.

As described above the features of the 8051 and the CC2430 is a mixed bag: on one hand the tight system integration and low price are attractive, on the other hand the legacy architecture might be a deterrent. The question is: can we abstract the aging architecture with modern abstractions and still benefit from the device, while maintaining ease of programming?

4.5 TinyOS 2.0 on 8051

TinyOS 2 is the successor to the highly successful sensor network operating system TinyOS. While TinyOS 2 is built on largely the same foundation as TinyOS 1, the emphasis has been shifted towards portability. Built on the same component model as TinyOS 1, version 2 adds a set of documents formalizing hardware abstraction interfaces - essentially a driver interface. The documents, denoted as TinyOS Enhancement Proposals (TEP), are intentionally implementation oblivious and can be seen as an interface contract stating optional and required functionality. In a nutshell writing a new platform for TinyOS 2 consists in providing the required interfaces with the functionality described in the appropriate TEP.

While the principle is simple there are a some practical hurdles to overcome for this to become reality. The use of contracts in prose opens the door to interpretation; simply understand-

ing the intention of the text is the first task of a new platform implementer. Next of course is decide how to best provide the described functionality on a particular platform—here looking at the code of other implementations is useful. Additionally other bits of practical information has not made it into a formal description.

Before implementing an actual platform, the first task we tackle is to interpret a subset of TEPs that we consider to be the core and provide description of our interpretation and implementation. As the TEPs are relatively new and the implementation of current platforms have evolved along with the TEP descriptions, we believe we are the first group to undertake this task. The findings are published in a technical report separate to this dissertation[58]. We use this to implement the interfaces described in Section 4.5.5.

4.5.1 TinyOS Tool Chain

TinyOS programs are written in a C dialect (NescC) that augments C with a component model. The TinyOS 2 tool chain reads the NescC source files and passes them to a pre-compiler that produces a C program which is passed to GCC (see Figure 4.2(a)). The major component in the TinyOS tool chain ”nescc” - a source to source pre processor reading nesC and generating C (in GCC dialect). Unfortunately gcc is not available for the 8051, given that NesC produces code in gcc-dialect this becomes a problem when attempting to use a different compiler. Furthermore expressing some of the features of the 8051 processor is commonly done by extending the ANSI-C language by types and keywords.

Apart from interpreting the TEPs the major hurdle in adapting TinyOS to the 8051 architecture lies in:

- Adapting the TinyOS tool chain. TinyOS is built using the GCC tool chain, but GCC is not available. The TinyOS tool chain is built using GNU Make and this system has to be integrated with the compiler.

  The compilers described above can all by operated from the command line, which makes it relatively straightforward to introduce the relevant make rules in the TinyOS tool chain.

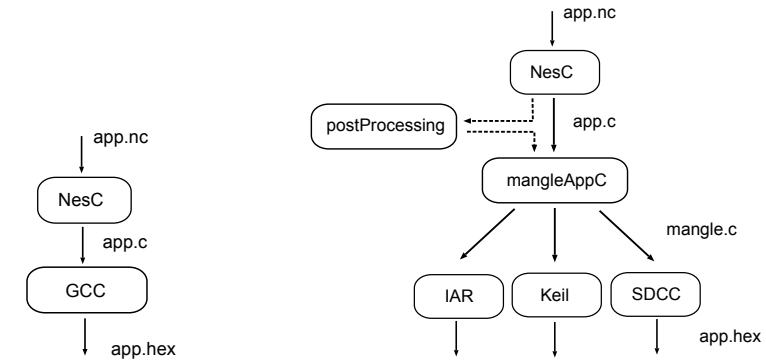
  By using the command line versions of the compilers we are also able to transparently introduce the Wine Windows translation layer[3]. In this way the tool flow can be compiled seamlessly other platforms than Windows despite the fact that the compilers are only available for Windows.

- Adapting the ANSI-C and GCC dialect to the 8051 variants of C.

  - Some of the features cannot be programmed using ANSI-C and the compilers often extend the C language with architecture specific extensions. The extensions are often very similar across compilers, but the syntax is often slightly different.
  - NesC relies in GCC to perform inlining. The code produced by NesC contains numerous superfluous function calls that can trivially be implemented, but unfortunately none of the described compilers support inline at the time of writing.

It turns out that the second problem is non trivial, and we solve it by automatically modifying the code at compile time, or code *mangling*.

[3] http://www.winehq.com



(a) TinyOS toolflow (b) 8051 toolflow

Figure 4.2 *(a) The unmodified TinyOS tool flow, (b) the modified 8051 tool chain, passing the code from nesC to mangleAppC and further to one of 3 compilers, possibly passing the code through an additional post processing step. We use this step to insert an a stand alone inline step.*

4.5.2 Mangling the Code

To solve these problems we introduce a simple C to C transformation step in the tool chain (see Figure 4.2(b)). TinyOS tool chain built using GNU Make, and introducing this step is a simple matter of inserting additional build dependencies in the makefile. Inspired by other implementations using the same strategy we call this the *mangling* step. This transformation turns GCC annotations into annotations for the compiler of choice. Each of the available compilers choose a different compilation strategy when targeting the limitations of the 8051 architecture and often differ in the C dialect. This solution allows a single source tree to be compiled with a number of compilers.

This strategy allows additional source-to-source transformation steps to be inserted seamlessly in the tool chain. On going research is currently looking into using such steps to improve performance, security and reliability. For us this is allows an elegant solution to the inline problem. By inserting the standalone *utah-inliner*[4], the code is inlined before being mangled. To our knowledge this make the tool chain the only 8051 enabled tool chain with reliable automatic inline.

This solution is far from ideal. As mentioned earlier the oddities of the 8051 is usually solved by extending the ANSI-C standard. This means that for the code to pass through nesC we need to either a) update nesC to accept the extended syntax or b) annotate the code in a way that passes through nesC. The first solution requires that each step that follows nesC is also modified to accommodate the extended syntax. We choose the later solution and implement a simple source to source transformation using Perl regular expressions.

4.5.3 Modification Details

The changes to the code can be put into two categories: i) the differences or absence of certain keywords in the target compiler and ii) the need to carry additional non ANSI-C compiler hints through the compile process all the way to target compiler.

[4]The tool was developed by researchers at the University of Utah specifically for TinyOS and slightly adapted to support the 8051 architecture. The tools are available, free of charge here. http://www.cs.utah.edu/ coop/research/tools/

For example consider the keyword *inline* part of the C99 standard. This keyword in not available in Keil and there is no equivalent, so our only option is to drop it. On the other hand annotating a function as an interrupt handler is expressed in GCC as *__attribute((interrupt(5)))*[5], while it is annotated with *interrupt 5* in Keil. Neither of the two are C99, but they are quite easy to transform.

Finally Keil uses some compiler hints that have no equivalent in GCC or C. Of these the most important are the ones relating to the 8051 memory model. For example the special function registers must be declare using a combined type and memory type specifier *sfr*.

We choose to implement the mangling as a simple Perl script that uses regular expressions to identify and modify the code. Regular expressions are easily used to do single line modifications, by substituting one pattern with another. While more elaborate code transformation could be required this has proven powerful enough for the examples above and more example like them. The 8051-oddities handled by the mangle script are:

- TinyOS (nesC) assumes glibc will be used and includes headers from the system locations. These headers are specific to glib C and are incompatible with the libraries provided by the compilers.
- nesC attempts to include system includes from GNU-libC, these are incompatible with the libraries provided by each compiler. To solve this we do two thing, first we provide dummy includes for NesC, later the dummies will be replaced by the compiler specific includes.
  - One notable example is the size of variable types. TinyOS includes system variable that define for example uint16_t to the appropriate C-type. This type is not the same in Keil and in GCC.
  - 64 bit integer (uint64_t) are not supported. Fortunately they are rarely used in the TinyOS code base and we simply drop the definition, resulting in a compile time error if a program should require 64 bit integers.
- GCC specific annotations are removed including (__attribute((packed)), #line, etc.)
  - Any inline annotations that remain are removed (if the inline tool was used the code has already been inlined).
- Any 8051 specific annotations are transformed to their final representation
  - "data" is a reserved keyword (it refers to a memory area). We replace data with _data
  - sfr, data, xdata, etc. are passed as __attribute annotations to variables that are later replaced for the specific compiler syntax.

    ```
    uint8_t volatile  RFIM      __attribute((sfrAT0x91));
    ```

    becomes (for Keil)

    ```
    sfr  RFIM = 0x91;
    ```

  - Interrupt definitions are changed from a GCC like syntax to a compiler specific syntax, note that the interrupt number is passed as part of the function name. This could be done more elegantly using parameters to the interrupt attribute.

[5]The syntax varies slightly from platform to platform

```
void __vector_2 (void) __attribute((interrupt,
                                    spontaneous, C))
```

becomes (for Keil)

```
void __vector_2(void) __interrupt(2)
```

- C99 automatic array length are not allowed in any of the C compilers we have tested. Converting this is beyond the scope of the simple regular expressions of the mangle script. We settle for simply detecting this language construct and aborting compilation.

While this solution has worked reliably with few problems during our development, the changes above are on the boundary on what is practical as a simple Perl script.

Pointers and Generic Components

One of the ways uses of generic components in TinyOS is to instantiate multiple component wrapping identical hardware blocks. Such blocks are usually defined by a few hardware register, that can be passed to the component when it is instantiated. Now, recall the discussion on pointers - how do we pass pointers, written in ANSI that refers to a very specific location in 8051-C?

NesC is an extension of C, and passing pointers in C is relatively straightforward, but the pointer model assumes a uniform von Neuman memory space. In the case of Harvard architectures this is not the case, but even so it rarely becomes a problem: the C-compiler handles the code memory space and the pointers written by the programmer refer to data memory. The 8051 architecture further complicates the matter, by further subdividing the memory. Consider the following example: declaring an integer and passing a pointer to this variable is now ambiguous.

```
new HplAtm128GeneralIOSlowPinP((uint8_t)&PORTG,
                               (uint8_t)&DDRG,
                               (uint8_t)&PING, 0) as G0);
```

If we were to do the same in case of the 8051 we would need to qualify to which memory area the pointers refer. Doing so is however is not covered in C. Our simply solution is simply omit using generic components in this way for the 8051 family.

At a later point one could imagine adding an annotation form that passes through NesC to the underlying compiler. In this way NescC can ignore the annotation as long as it is preserved through the remainder of the tool chain.

4.5.4 8051 Platform Family

The directory structure of TinyOS follows the composition mindset. A platform is composed of a set of chips and multiple platforms can reuse chips. We wish to continue this division, but further we have a set of chips that are them selves part of a family. The CC2430 is an 8051 variant, but the core of the CC2430 is shared among other Texas Instrument chips.

We choose a simple strategy by simply creating directories for each of these that inherit from each other, but are not organized to reflect this (see Figure 4.3).

Timer.h

```
mcs51
  tos/chips
mcs51
nRF24E1
cc2430
  radio
  timers
  adc
  tos/platforms
nano
cc2430em
nRF24E1_EVKIT
```

Figure 4.3 *TinyOS 8051 directory structure. Current CC2430 based platforms include the TI development kit cc2430em and sensinode Nano. The cc2430 inherits common functionality from the mcs51 directory. TH nRF24E1 chip and nRF24E1_EVKIT platforms are currently under development.*

4.5.5 TinyOS Components

TinyOS 2 in itself has been structured to allow code to be reused across platforms. TinyOS is built using the concept of components that use and provide interfaces. TinyOS provides a set of system components (e.g. initialization, scheduler, etc.) a set of library components (e.g. random number generator, queue, etc.) and a set of platform specific components. A new platform can reuse the provided libraries and system component by adding abstractions for the underlying hardware components.

In the following we will describe how we build our platform using platform independent interfaces. We focus on the following hardware components:

Radio We export the radio using a straightforward *SimpleMac* interface. This interface is well suited for the 802.15.4 packet-based radios of the CC2430. It allows to send and receive packets, and set various 802.15.4 parameters as well as duty cycling the radio. Note that we depart from the Active Message abstraction promoted by the TinyOS 2.0 core working group. Our SimpleMac implementation supports simple packet transmission, but does not provide routing, or retransmission. Implementing Active Messages is future work.

Flash We export the flash using the *SimpleFlash* interface that allows to read and write an array of bytes, as well as delete a page from flash. Note that this interface is much simpler than the abstractions promoted by the TinyOS 2.0 core working group (volumes, logging, large and small objects). We adopted this simple interface because it fits the needs of our data acquisition application. Implementing the core abstractions as defined in TEP103[31] is future work.

Timer The timers are exported using the generic TinyOS Timer interfaces *Alarm* and *Counter*. These two interfaces give applications access to hardware counters and allows the use of the TinyOS components to extend the timer width from 16 bit to 32 bit. Note that on the pre-release CC2430 chips we used for our experiments, timers do not work properly[6].

ADC The Analog-to-Digital Converter is accessed through the core *Read* interface that allows

[6]The timers miss events once in a while. This error is documented on a ChipCon errata, which is not publically available.

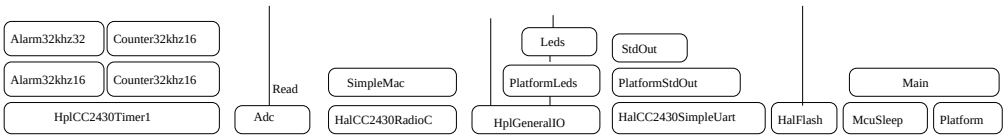


Figure 4.4 *TinyOS Components*

to read a single value. In order to read multiple values, an application must issue multiple read calls or use DMA transfers.

Pins The General IO pins are exported through the core *GeneralIO* interface, that allows to set or clear a pin, make it an input or an output.

UART The UART is exported using the core *SerialByteComm* interface (that sends and receives single bytes from the UART) and *StdOut* interfaces (that provides a `printf`-like abstraction on top of SerialByteComm.

Radio The radio is exported using the straightforward *SimpleMac* interface. This interface is well suited for 802.15.4 style packet based radios.

Timer The timers are exported using the generic TinyOS Timer interfaces *Alarm* and *Counter*. These two interfaces gives applications direct access to hardware counters and allows the use of the TinyOS components to extend the timer width from 16 bit to 32 bit.

ADC *Read*

Pins *GeneralIO*

UART *SerialByteComm*, *StdOut*

SimpleMac

We abstract the radio using an interface well suited for 802.15.4 style packet based radios (such as the CC2430, Motorola MC13192, etc.). This allows simple access to packet transmission, but does not provide routing, retransmission, etc.

Advances with system on a chip design (SoC) has provided cheap, feature rich devices well suited in sensor networks.

4.6 Experimental Results

We create a simple experiment to enlighten some of the parameters that we have discussed in this chapter. The benchmark consists a compression test application using the experimental data from the Hogthrob pilot experiment[71]. The test application reads data from a serial port into a buffer, compresses it and sends it back. The data is transmitted from a PC in 256 byte chunks, in total 4 KiB data is transferred and compressed. The PC starts the compression by sending a character over the UART that signals the mote to begin compression, upon completion the mote sends a character back. The PC records the time duration between the transmission and reception of these characters, not counting the UART transmission of the data chunks.

```
interface SimpleMac
{
command error_t sendPacket(packet_t *packet);
event void sendPacketDone(packet_t *packet, error_t result);

event packet_t *receivedPacket(packet_t *packet);


command error_t setChannel(uint8_t channel);
command error_t setTransmitPower(uint8_t power);

command error_t setAddress(mac_addr_t *addr);
command const mac_addr_t * getAddress();
command const ieee_mac_addr_t * getExtAddress();

command error_t rxEnable();
command error_t rxDisable();

command error_t addressFilterEnable();
command error_t addressFilterDisable();

command error_t setPanAddress(mac_addr_t *addr);
command const mac_addr_t * getPanAddress();
}
```

Figure 4.5 *SimpleMac*

The compression test application is built using a plugin algorithm, allowing us to replace the algorithm at compile time. This allows us to use two compression methods that differ mainly in their computing intensity and their memory access pattern:

Simple Simple lossy algorithm. This algorithm uses a simple heuristics to drop data, compress them using a simple run length encoding scheme. This means that the data handling

LZ77 Well known loss-less algorithm. One of the features or the algorithm is to constantly maintain a *window* of recent data. This window is accessed repeatedly through the compression process.

The test setup allows us to look into some of the parameters regarding code generation and the performance of the executed programs. For the experiment we used a ChipCon development kit with a pre release CC2430 (chip revision 0). The chip had some bugs and it is likely that some of the test results will differ in later versions, further we had problems with the code being unstable using other compilers than Keil, and we only report runtime using Keil. Further the recent version 2.7 of the SDCC contains inline, but the compiler simply crashed (segfault) when attempting to compile the code without handling inline separately. We used the following compiler versions:

Keil PK51 8.09 (μVision version 3.51)

SDCC 2.7.0

IAR 8051 7.30B

The compression test transfers the data as 16 chucks of 256 bytes and compresses 4 KiB in total.

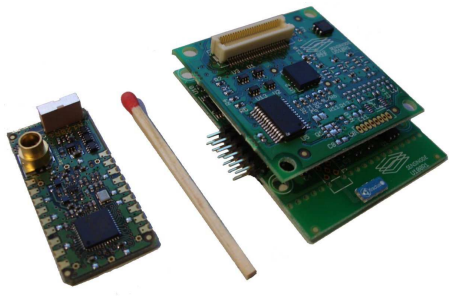


Figure 4.6 *The Sensinode Nano (left) and Micro.4 (right) platforms compared to a match. The Micro.4 is shown with an accelerometer board attached.*

4.6.1 Platforms

We use two platforms from Sensinode[7] for our tests: one is the Nano featuring the CC2430 SoC and the other was the Micro.4 featuring the highly popular MSP430 (see Figure 4.6). We use the Sensinode Micro.4 as our comparison as it is very similar to other popular sensor network motes (see Chapter 2).

The Micro.4 features the Texas Instruments CC2420 radio and an additional flash. The external flash and radio are access through a shared SPI bus, controlled by a simple bus arbiter. The Micro.4 is designed in a stackable fashion allowing sensor boards to be added easily.

The Nano platform is intended to be added to an existing design, and is designed with solder pads on the edge. The pads can be fitted with pins allowing the module to fit into regular chip-sockets. Sock a socket could for example contain a sensor board, and such boards are available from Sensinode.

4.6.2 Code Size

We evaluate the code size of the compiled programs using GCC on the Micro.4 (MSP430) as a reference point. In case of the CC2430 we vary the compiler (Keil, Iar, SDCC) and we compile with and without the separate stand alone inline tool. We had problems compiling the code using Iar and SDCC without this tool. The cause is likely that the inline tool also contains a cleanup step—without it the auto generated code from nesC seems to trip bugs in the compilers. We have not investigated the cause of these bugs further. The results are shown in Figure 4.6.2.

Despite the fact that the architectures are substantially different and the tools involved in the compilation process the code sizes are remarkably similar. It has been claimed that the CISC style 8051 would generate far larger programs than the RISC style MSP430, this is not supported by our findings, on the contrary we do not show a significant difference. Similarly one would expect inlining to have a greater effect on code size, but again this does not seem to influence the code size of our benchmark substantially.

Next we will look at the run time performance, while the codesize may be close with and without inlining—the run time might still differ.

[7] http://www.sensinode.com

Platform, compiler	Code (KiB)	RAM (KiB)
MSP430, GCC	4.2	6.3
CC2430, Keil, inline	5.0	6.5
CC2430, Keil, no inline	5.1	6.5
CC2430, SDCC, inline	5.9	6.4
CC2430, SDCC, no inline	5.9	6.4
CC2430, IAR, inline	4.0	10.3
CC2430, IAR, no inline	4.6	10.3

Figure 4.7 *Code size and memory usage reported by each compiler of the compression test using LZ77 for the MSP430/CC2430 using a set of different compilers. We compiled the program with and without the stand alone inline tool, in the cases where no separate inlining is performed the inline keyword is simply removed. For all compilers we use the large memory model that places stack frames in xdata memory, this is a safe choice, but results in lower performance and higher code size caused by the increased data handling.*

Platform	Current
MSP430 (1MHz)	1 mA
MSP430 (4MHz)	4 mA
CC2430 (16MHz)	5,8 mA
CC2430 (32MHz)	9.8 mA

(a) Average power consumption

Platform	LZ77 (s)	simple (s)
MSP430 (1 MHz)	95.58	0.70
MSP430 (4 MHz)	23.90	
CC2430 (16 MHz)	69.25	
CC2430 (32 MHz)	28.23	0.12
CC2430 (32 MHz, no inline)	34.63	0.12

(b) Compression test runtime

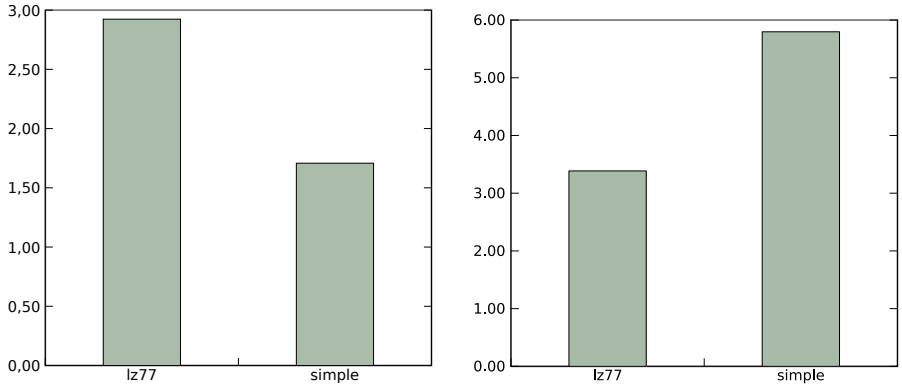
Figure 4.8 *a) Average power consumption measured during the experiment b) runtime measurements for the MSP430 and CC2430 respectively*

4.6.3 Power Consumption and Run Time

Timing is recorded using a PC while a multimeter is attached to the mote to record the energy consumption. The run time is averaged over 10 runs using 10 different sets of 4 KiB data. We use the Keil compiler inline for these experiments, in the 32 MHz case we further run the experiment without the inline step. The results are shown in Figure 4.8 and the ratio between the Micro.4 and CC2430 are show in Figure 4.9.

In Figure 4.8(b) we notice that not surprisingly the execution time is proportional to the clock frequency for both motes. We also note a significant performance advantage is obtained by using the separate inline step, this is likely caused by the way nesC generates code—it assumes that the compiler will perform the inline process removing function calls that can trivially be eliminated. The figure also shows that the 32 MHz CC2430 is faster than the 1 MHz Micro.4 in both the simple and the LZ77 algorithm, but notice by comparing the ratio in Figure 4.9(b) that while the CC2430 is 6 times faster for the simple benchmark it is only 3 times faster for the LZ77. This difference is likely due to the difference in memory access pattern of the two applications: Both compression algorithms store buffers in xdata, but lx77 uses 2nd buffer (window) during compression. In this way lz77 pays the xdata tax more than once per sample.

The energy consumption of the CC2430 is considerably higher than the Micro.4 as shown in Figure 4.8(a), but looking at Figure 4.9(a) we notice that in the simple case the reduced run time compensates and energy consumption of the two is almost equal. However in the LZ77 case



(a) Energy ratio (CC2430/M.4) - CC2430 consumes most

(b) Runtime ratio (M.4/CC2430) - CC2430 is faster

Figure 4.9 *Performance ratio between the CC2430 (32 MHz) and the Micro.4 (1 MHz) when running the LZ77 compression a) Energy consumption ratio, notice that the ratio is different for the two application: for lz77 the cc2430 consumes more than 3 times the energy for the same task than the Micro.4, but for the simple compression they are almost equal. b) Runtime ratio notice that the CC2430 is between 2.5 and 6 times faster than the Micro.4.*

the CC2430 consumes considerably higher amounts of energy to complete the same task.

4.6.4 Observations

We have conducted a few simple experiments that illustrate some of the advantages and disadvantages of the 8051 platform in general and the CC2430 in particular. We make the following observations:

- The energy consumption of the CC2430 is considerably higher then that of the Micro.4. The high energy consumption is like caused by the high clock rate. Regardless of the cause the high energy consumption combined with the performance bottlenecks makes for an unattractive combination for computing intensive tasks.

  The observed values are close to the ones reported in the data sheets and it seems likely that they are not caused by bugs in the pre release chip.

- The 32 MHz mode of the CC2430 is only required when operating the radio. We notice that the power consumption of the CC2430 in 16 MHz mode approaches that of the Micro.4 in 4 MHz mode. For us this means that for low computing intensive applications the CC2430 still features an attractive energy consumption profile.

- In our examples the code size for MSP430 using the GCC compiler and the CC2430 using the Keil compiler is very similar. This contradicts conventional wisdom that the code size should be significantly different using the 8051 architecture. In our experiments the code size is very similar.

4.7 Limitations

In this chapter we have presented our TinyOS port to the 8051 based CC2430. We conducted experiments using the platform and we are confident that this implementation can lay the foundation for continued use of the CC2430 platform within the TinyOS community. While in general the results are reassuring in many ways, the platform is far from finished and in the following sections we will cover the most urgent issues that must be addressed before this platform can be rolled out in larger scale experiments.

- We did not go into the implementation in great detail, but there are still some features that are not implemented or not implemented according to the TEP's. In particular we point to the following:

  AM Radio stack Here we have used the *SimpleMac* interface that is easy to port and has been the basis of several 802.15.4 radios. This interface is however not the accepted standard within TinyOS. Implementing the AM Radio stack will enable existing applications to transition to this platform, as well as encourage users familiar with this interface.

  The radio of the CC2430 is very similar to the radios that currently provide the AM abstraction (such as the CC2420) and we see no immediate obstacles in doing so. One strategy is simply to port the CC2420 stack, a task that is currently underway.

  Precise Timers In this work we exported the timers using the appropriate interfaces, but the precision was not as advertised by the interface. The reason was a bug in the pre release silicon revision of the chip that was available to us. The timers miss events every now and then—for an application expecting periodic events missing an event will double the period. The workaround used here was to simply run the timer at a much higher rate resulting in a shorter time between missed events.

  As more mature chip revisions become available changing the timers to work at the advertised rate is essential.

  Remaining chip features There are still a number of essential chip features that are unsupported by our framework. The most important is sleep modes. Initially we had some problems implementing the sleep modes on the pre release chips, however it should be a trivial task for mature chip versions.

  Other features such random number generator, etc. are also important.

- The implementation consists of more than 200,000 lines of code, simply weeding out the bugs is a daunting tasks. Generally, if researcher are to consider this project attractive and invest time in building applications using it, it is essential to build confidence in techniques and the code base.

  Further experiments In both cases more experimentation is required. In this work we only carried out simple desktop experiments, the next step is to increase the scale an perhaps carry out field experiments. It is unclear whether the Hogthrob project will be able to carry out further experiments, so an other way to attract interested parties is through the TinyOS community.

  We have share the code and made it available through the following website, it is our hope and belief that this will attract attention to the project.

`http://www.tinyos8051wg.net`

Mangling Our experiments have uncovered few problems in the simple source to source transformation written in Perl that we have presented. The question is if this simple strategy will be enough or if more elaborate tools are required. In particular if a similar technique is going to be used to enable a new set of compilers for TinyOS. One solution might be to rewrite the transformations using a framework built for this purpose such as CIL[8].

- One of the most peculiar features of the 8051 is the use of banked memory. In this work we disregarded the use of banked memory, which limits the available code memory on the 128 KiB version of the CC2430, but simplifies the programming using C. Fortunately 55 KiB of code memory is available without this technique—a sufficient starting point.

  Moving beyond this point requires explicit annotating code and variables and possibly hand tuning the location of some objects. In Keil objects that are located in the banked area are addressed using the tree-byte generic pointer construct causing an overhead, and calling functions in an other bank imposes an overhead in terms of the bank switching.

4.8 Conclusion

In this chapter we have demonstrated our implementation of TinyOS 2 for 8051:

With our proof of concept implementation we have shown that the abstraction level of TinyOS 2 was sufficient to cover the 8051 platform that differs substantially from the platforms on which TinyOS was first implemented.

We have shown that our port of TinyOS is flexible enough to allow seamless migration of TinyOS applications to the 8051 platform. We put emphasis on the adaption required to TinyOS in order to support the 8051 in general, and the specific features of the Texas Instruments CC2430.

This proof of concept clearly shows the feasibility of the task. We find no immediate obstacles in the system architecture of the 8051 that prevents it from becoming popular within the TinyOS community. We find that the hardware abstraction architecture of TinyOS 2 is general enough to allow relatively low level applications to easily take advantage of the advantages provided by this system on a chip device.

- The modifications to TinyOS platform was mainly language constructs—some hardware specific functions required the use of non ANSI-C constructs. In case of the CC2430 the peripheral units are a good match to the abstraction interfaces found in TinyOS.

  As such the challenge in porting TinyOS further 8051 based devices lies in the peripheral units. Other devices may provide hardware units that are less closely matched to the TinyOS abstractions.

  One such example is the Nordic Semiconductor nRF24E1. This devices resembles earlier 8051 devices more than it resembles for example the CC2430 or MSP430. The nRF24E1

[8] http://sourceforge.net/projects/cil

devices for example limits the possible timer pre-scaler (clock divider) values, that may or may not be a problem.

We are currently in the process of adding a nRF24E1 port to our suit of platforms.

- Beck et. al[6] concludes that the coupling of TinyOS (NesC) and GCC hinders TinyOS in it's ability to span new architectures. We do share this view, on the contrary we consider it to be an advantage to maintain a single C dialect within TinyOS, that can later be automatically transformed to a specific compiler (covered in Future Work below).

  Beck et. al[6] further conclude that without substantial changes in TinyOS, it will remain challenging to port TinyOS to non GCC based platforms. We do not share this conclusion, and point to the fact that creating an operating system that abstracts different hardware architectures is in it self a challenging task. Doing so in a compiler agnostic fashion and covering several language dialects only adds to the equation.

- To our knowledge we are the first group outside besides the authors of the TEP's to take on the task of interpreting the TEPs into a concrete platform. The findings of the interpretation of the TEPs is published separately as a technical report[58].

We believe that this implementation is superior to earlier attempts for previous versions of TinyOS in a number of ways:

- By taking advantage of the recent TinyOS 2 hardware abstraction architecture lowering the effort required move existing TinyOS applications to this platform.

- This port is considerably more complete then earlier attempts for TinyOS 1.x[82] and TinyOS 2.x[6], with a strong focus on portability, support for 3 popular compilers, integrated inline support, support for several platforms, etc.

- To our knowledge we are the first to embedded 8051 operating system to integrate inline in the tool chain. In our case the way nesC generates code, the absence of inlining result in a significant performance penalty for the platform. The stand alone inline tool, developed at the University of Utah, solves this problem.

- To our knowledge we are the first group to provide an operating system for the 8051 platform providing both cross platform development (Windows, Linux, and Unix clones) and compilation process supporting multiple compilers.

- We have attempted to mimic some of the surroundings that have made TinyOS itself popular. The code is released on a website[9] with a community forum and a rich documentation. We have formed a TinyOS 2 working group, that will hopefully attract interested parties

  Only time will tell if this attempt to gain a higher level of popularity than earlier attempts for previous versions of TinyOS.

  It is our hope, that the above advantages, will build confidence in TinyOS for 8051 and inspire others to build on our work and promote the 8051 platform as a first class citizen in the TinyOS arena.

[9] http://www.tinyos8051wg.net

4.8.1 Contributions

In summary our contributions regarding portability are:

A highly compatible port of TinyOS 2 for the 8051 Our port for the 8051 and CC2430 platform is able to serve as a basis for further development using these platforms. The port can equally serve as a platform for other 8051 based devices.

We claim that this is a major step in enabling TinyOS for system on a chip based devices.

Comparison of the CC2430 and Micro.4 platforms We have compared the CC2430 platform to the Sensinode Micro.4 featuring the popular MSP430 microprocessor, with focus on some of the architectural differences that may impact the use within sensor network applications (code size, memory usage, execution time).

The comparison presented in this chapter complements the results presented in Chapter 3, further building the case for the CC2430 and 8051 in sensor network applications.

Lessons for portable sensor network operating systems The challenges faced here are not limited to the 8051 based platforms or TinyOS. The solutions are applicable to a larger set of devices and software support systems.

In particular the techniques for cross compiling a single source tree for multiple may be interesting to other systems facing similar challenges. Similarly other systems may find insights in our specific recommendations for new TinyOS 2 platforms[58].

In the case of TinyOS integrating the GCC dependent tool chain with other compilers can be interesting to other TinyOS platforms.

4.8.2 Lessons Learned and Future Work

We employed the use of a source to source transformation in order to adapt the source tree from a common dialect (GCC) to the dialects of 3 different compilers. The purpose of this transformation was twofold: i) to transform compiler hints from their GCC representation to the specific compiler representation (interrupts, inline, etc.) and ii) to allow the use of non-ANSI extensions required by the 8051 architecture (bit variables, special function registers, etc.)

These transformations were specific to the 8051 architecture, but the use a single source tree with a number of different compilers is generally interesting. For example IAR also provides an MSP430 compiler, TinyOS is currently not able to take advantage of this compiler, for reasons very similar to the case presented here.

- We believe that introducing a similar source to source transformation step to TinyOS in general would be beneficial to all platforms. As in our case this transformation step would be introduced just prior to the compiler in the tool chain after nesC and any addition post nesC steps. By fixing the internals to one C dialect each of the intermediate steps can be limited to handle only the one dialect.

  In this work a simple Perl script was sufficient, it is an open question whether this is sufficient in the general case.

We compared a traditional two chip solution to a highly integrated system on a chip device. We saw a significant efficiency benefit from the integration of the radio and MCU. For the CC2430 the picture becomes more blurred in relation to CPU performance, while it is significantly faster than the Micro.4 it is also consumes significantly more power, most likely because of the high clock rate.

- It is an open question which applications will benefit from this trade off, but clearly the low price and high radio efficiency is attractive for applications that do only require short periods of computation.

  In this work we did not investigate the performance gain obtained as a result of the high integration—quantifying this performance gain is future work.

- Our experiments show no clear winner: the CC2430 or the Micro.4. We have shown that for a class of applications with low focus on computing intensity the CC2430 may very well be attractive.

  – In this work we focused on the CC2430, which is a likely extension to applications using the popular MSP430/CC2420 combination. However obtaining results from the nRF24E1 or other 8051 based clones will be interesting. The nRF24E1 datasheet[10] reports the energy consumption to 3 mA running at 16 MHz, which might make it more attractive to more computing intensive applications.

- Comparing this device to other devices that feature lower clock rates would shed more light in a clock to clock comparison. The Nordic Semiconductor nRF24E1 is an example of such a device featuring clock rates from 4 to 20 MHz. We are currently in the process of using the general framework described here to port TinyOS 2 to this platform.

- In this work we noted a significant performance increase when introducing inlining. The high increase stems in part from the nesC generated code which introduces superfluous function call, but also from the high overhead imposed by the 8051 architecture. As mentioned because stack space is limited the common solution chosen by most compilers is to allocate a static stack frame in the slow access data memory imposing an overhead on function calls.

  An alternative solution is to use a traditional strategy that allocates stack frames dynamically. In the common case the stack is generally unbounded making this a risky choice, but in case of TinyOS the we are able to use whole program analysis that enables bounding the stack size[73, 91]. In this way the safety of the program execution can still be guaranteed without paying the overhead of the pre allocated stack frames

  Investigating whether relying on a traditional stack approach for TinyOS is feasible with the limited stack size of the 8051 and the possible performance gains combining these two approaches is future work. One way could be to combine the `--stack-auto` feature of the SDCC compiler[11] with the source code stack analysis `stack-estimator`[73].

[10] http://www.nordicsemi.no
[11] http://sdcc.sourceforge.net

CHAPTER 5

Conclusion

In this dissertation we have addressed two important issues that we faced within the Hogthrob project. The Hogthrob project focuses on sensor networks for sow monitoring and sets difficult goals for performance and price that current solutions are unable to meet: a few euros in price and two year lifespan. Tackling these problems lead us to consider a much wider design space than previously seen in sensor network applications. By opening the playing field to *specialized* mote design as opposed to *generic* motes we are able to consider attractive solutions, for example system on a chip devices and hardware accelerators. However moving to these devices also complicates development. Chip design is a research topic in its own right, and understanding performance of a device in relation to an application prior to building it is difficult. Building chips in order to gain familiarity with the possibilities, advantages and limitations is time consuming and expensive, and far beyond the capabilities of the Hogthrob project.

To move towards these challenges, we took two approaches. In both cases we employed a practical approach and implemented proof of concept implementations to support our claims. First we developed the new vector based performance characteristics (Chapter 3), this method allows us to predict the performance of a specialized mote without having to build it. Second we fixed a point in design space by looking at the 8051 based platforms in general and the Texas Instruments CC2430 in particular (Chapter 4). In order to gain access to this platform, we ported the highly popular TinyOS 2 sensor network operating system, claiming to provide framework to abstract architectural differences into platform independent interfaces using the hardware abstraction architecture. With this foundation we compared the CC2430 to a popular architecture used within the sensor network community, the MSP430.

With these advancements we claim to have made significant strides towards enabling specialized sensor mote design. Our motivation was based on a concrete example, in this way we focused on a set of specific parameters. The methods we have developed are, however, generally applicable. In our case we need to be able to estimate performance *a priori* as part of a design process, but this approach may simply be part of a mote selection process. Similarly we have chosen to port TinyOS to a particular platform, which enables other TinyOS application builders to consider this platform, but more generally we have provided a strategy for other platform implementers hesitating to explore unfamiliar platforms. We did this first in part by the problems described here and by providing an interpretation of the TEP documents published separately in a technical report [58]. Further we took part in the design of a compact FPGA based development platform enabling *in situ* experiments using hardware accelerators, the design considerations are have been published previously[57, 102] and a manual is published separately as a technical report[59].

The needs of the Hogthrob project were very specific, but it is our belief that for certain

classes of sensor network applications share this push for lower energy consumption, smaller size, and lower price. We claim that our solutions are directly applicable to such classes of applications.

In this dissertation we attack two key issues in building specialized sensor networks: portability and performance evaluation.

5.1 Performance Evaluation

Our thesis statement concerning performance claimed that current methods were insufficient to allow us to compare performance across motes.
We have built a methodology that allows us to objectively compare motes and application workloads. The application is different that earlier approaches in 3 ways: i) in the ability to give insights for real workloads as opposed to generic workloads, ii) in their ability to cover and entire mote into account as opposed to a single subsystem and iii) unable to support a prototyping approach by estimating the performance of an application on a target mote without having to implement the application on that mote and carry out an experiment. We claim that our vector based methodology meet all three questions challenges:

In this dissertation we show the viability of this methodology by implementing it in TinyOS 2 and carrying out experiments using two commercial platforms: The Sensinode Micro.4 and the Texas Instruments CC2430.

- The method allows real application to be used in two important ways: i) it extract a workload description from a real application instead of relying on a manual description of a workload and ii) this workload can be collected from a mote running in the field. This eliminates the uncertainties involved in attempting to create an estimation using a manual workload description or a simulation with a model of the environment.
- The mote vector describes an entire mote, and the performance evaluation uses all of these components to describe the behavior of a mote. The mote vector is constructed using a benchmark, that allows objective comparison across motes. This strategy is different than attempting to capture all the information of a benchmark in a single number.
- The vector based approach allows the application vector be collected on one mote and estimated on an other mote if mote vectors are available for both motes. In this way one mote can serve as a prototype through the development of a target mote. In this dissertation we have used this method to estimate performance between two exiting motes, but the method can potentially be used to estimate performance on a non existing mote before it is built by estimating the entries of the mote vector.

With this method we are equipped to answer the overall question of the Hogthrob: will this mote run for tow years without depleting the batteries? We are confident that this methodology is a significant step in enabling application specific design.

5.1.1 Contributions

In summary our contributions are:

A new methodology for sensor mote benchmarking We have proposed a method based on the principle of application specific benchmarking, describing a mote and an application using vector quantities. Combining the two yields a performance estimate.

An experimental verification of our methodology We have implemented the method in TinyOS as a logging layer capturing the behavior of any application. This layer is publicly available. We used the implementation to conduct experiments building the case for our methodology.

A quantitative comparison of two sensor network motes Using this method we compared two prominent sensor network platforms: the CC2430 and the Sensinode Micro.4.

5.1.2 Future Work

In this dissertation we have shown that the vector based methodology is viable. We have conducted experiments using two commercial sensor motes. The limitations of this methodology are, however not well understood:

- We have shown the viability of this methodology using TinyOS and two motes representing distinct points in the sensor network design space. Much more experimental work is required to learn the possibilities an limitations of this method.

  In this work we focused on a single operating system (TinyOS) and a few motes. Expanding parameter space is a next step, comparing more sensor mote hardware and mote operating systems.

- The algorithm can potentially expand time frame of collected traces by orders of magnitudes compared to current practices. Expanding the experiments time frame to days or months will uncover the true potential of this approach.

- The precision and the sources of inaccuracies are not exhaustively examined. This includes the impact of our CPU model, the use of an architecture matrix to model shared resources and more.

5.2 TinyOS on 8051

Our thesis statement concerning portability we claim that the recent TinyOS 2 architecture advertised as highly portable requires experimentation. We have built a proof of concept platform for TinyOS 2 on the 8051 and CC2430. We tested the claim that TinyOS 2 operating system was built using portable abstractions that allows it to span very different architectures, and provided the ground for building this architecture as a first class citizen within TinyOS.
We took a fixed point in design space by looking at the 8051 based platforms and the CC2430 in general. This enabled us to quickly get access to a highly integrated system on a chip device, that will provide insights for further development in the system on chip arena.

- We find that TinyOS 2 is well suited to support the 8051 and CC2430 architectures, and that the abstractions are general enough to this cover this platform. This counters the findings by Beck et. al[6]

- We have emphasized the use of TinyOS 2 standard interfaces allowing applications to move easily to this platform. This implementation will provide a new path of development for TinyOS into the system-on-a-chip domain, by enabling application builders to take advantage of the CC2430, but equally for others to build platforms based on this work.

- To our knowledge we are the first outside of the TinyOS core to interpret the TEP documents and to implement an platform based on this interpretation. Our interpretation is published separately as a technical report[58]. We believe this interpretation is generally interesting to TinyOS 2 platform implementers as well as other attempting to create portable abstractions for sensor network operating systems.

- We claim that this port of TinyOS is superior to earlier attempt at bringing TinyOS to the 8051. It is more complete, features performance enhancements (inline) and builds on the new TinyOS 2 standards, that enable easy platform migration. In addition we have attempted to mimic the community spirit that has made TinyOS itself popular, by creating a public website. It is our hope that this will attract attention to further develop this platform:

  `http://www.tinyos8051wg.net`

5.2.1 Contributions

In summary our contributions are:

A highly compatible port of TinyOS 2 for the 8051 Our port of TinyOS 2 for 8051 and CC2430 will enable application builders to take advantage of this new platform, making the advantages intrinsic to system-on-a-chip design to a wider audience.

Comparison of the CC2430 and Micro.4 platforms We have complemented our comparison using the vector based methodology with a comparison of some of the impacts of the architectural differences between these two motes.

Lessons for portable sensor network operating systems In this work we have presented the challenges related to integrating TinyOS and a set of compilers. We believe the solutions presented here are applicable to other platforms withing TinyOS as well as other sensor network operating systems facing similar challenges. This includes our overview of a TinyOS 2 platform published separately[58].

We believe that portability is emerging as an important topic within sensor networks. Here we have discussed the importance of portability of TinyOS—however as we mentioned in Section 2.3 we are seeing a general trend towards portability.

While we claim our work is a major step forward for sensor networks and TinyOS, some remains before reaping the fruits of our labor.

5.2.2 Future Work

In this dissertation we have taken a practical approach for creating portable sensor network operating systems. We believe this work can be expanded to accommodate a much larger set of variables:

- We have presented our solutions for the 8051 platforms. This implementation can be integrated more tightly with the main tree of TinyOS, allowing other platforms to benefit. A first step would be to implement TinyOS for other similar platforms[1]. Second, this work can provide multi-compiler support for existing platforms. Thirdly, it can provide support for TinyOS moving to new and unexplored platforms.
- This platform still suffers from some deficiencies. Completing these and building larger test cases is a next step in building confidence in the code base.
- Investigating how our solutions relate to other sensor network operating systems is an interesting continuation.

5.3 Perspectives of the Hogthrob Project

The Hogthrob project set out collectively to explore the use of sensor network technology in sow monitoring. We have presented some of our efforts within this project, but numerous challenges remain with in the project.

- We presented the HogthrobV0 prototype platform. This platform allows practical hardware/software co-design. We believe that the full potential of this platform has not yet been explored.
  - Investigating practical solutions for hardware software co design within the field of sensor networks.
  - This platform allows unsurpassed flexibility in experiments and providing *in situ* experiments of hardware accelerators, both future work.
- The experiments that we have not discussed at length in this work did not make it past the prototype stage. The were designed to operate for a month at a time, which was sufficient to collect useful data, but far from the two year lifetime goal. It is still not clear how a solution that meets the requirements should look. Our work and the work of the Hogthrob project in general have built solutions that will enable future researcher to answer this and similar questions, but a solution for the Hogthrob project does not exist.
  - Further experiments are required to solve the original challenges: implementing a detection model and verify that a solution meeting the requirements is indeed possible.
- In the introduction we briefly mentioned some of the choices we made for the pilot experiments. One of the choices we made was to use some of the cheap and commercially available 2.4 GHz radios—this type of radio has remained fixed since onset of the project. Since then a new standard has emerged, that may be more suited to this project: the use of near field communication, or the NFC standard[2]. In short NFC is an extension of RFID, short (10 cm) range, low bandwidth. It can operate in a passive mode requiring the device

[1] This work is currently underway as a student project implementing TinyOS 2 for the Nordic Semiconductor nRF24E1

[2] http://www.nfc-forum.org

to enter an electric field that drives the radio without requiring a power source on the device.

The use of NFC technology would push the energy burden of driving the communication to the infrastructure alleviating the device of the major energy consumer. In the context of Hogthrob the sows are already with RFID tags with readers installed in the feeding stations, making this an attractive choice.

After completing my Ph.D. I will join an NFC chip provider, and exploring this path is interesting continuation.

- We have discussed our efforts to move to the 8051 based CC2430 platform. The full potential of this platform in the context of the Hogthrob project has not been explored, is this platform sufficiently effective to meet the requirements?

  An immediate followup to this is to explore the use of hardware accelerators. In Chapter 2 we briefly described how the HogthrobV0 platform is setup as with an equivalent functionality as the CC2430, by using an FPGA with and 8051 compatible CPU and a second radio chip. Exploring the benefits of application specific hardware enhancements is future work.

- The potential of the HogthrobV0 platform has not been explored. This platform is ideal for studying the impact of hardware software co-design, combined with *in situ* experiments. This is an ongoing research effort in the DTU group of the Hogthrob project.

- In this work we have not discussed any animal science problems related to the Hogthrob project. This includes insights, modeling and processing required to detect events such as the onset and heat in sows based on the collected data. This investigation and modeling is part of the research effort at the LIFE group of the Hogthrob project, and the subject of the recent Ph.D. dissertation by Cécile Cornou[19]. Cécile Cornuo will continue some of this work during a post doc position at LIFE.

5.4 Summary

In this dissertation we have presented our solutions to two practical problems that we faced within the Hogthrob project. In the Hogthrob project we focus on sensor networks for sow monitoring. This application set very hard requirements on price and performance, that lead us to consider a much wider design space than previously seen in sensor network applications. In particular we focused on building an application specific sensor mote as opposed to a generic sensor mote and the use of system-on-a-chip (SoC) devices. Moving in this direction, however, raises a number of challenges. In this dissertation we attacked two key problems:

First we require the ability to evaluate the performance of a mote before it is built. We consider current sensor network mote evaluation techniques to be insufficient and we developed a new method based on earlier work by Seltzer et. al. This method uses a set of vectors to describe a mote an application. Combining these yields an estimate of run time and energy consumption on a given mote. This allows one mote to serve as a prototype mote while we are developing a target mote, while we are able to estimate performance on the target.

Second building a SoC is expensive and time consuming. As a result we took a fixed point in design space and implemented the popular sensor network operating system TinyOS on the MCS51 based CC2430 from Texas Instruments. This chip is representative of system on a chip

devices in general and of the class MCS51 based SoC devices that are currently emerging. The implementation has been made available through a public website, that we hope will attract interested parties to participate in the further development.

We claim that these are both significant advances in the current state of the sensor network research field.

Bibliography

[1] ACM. *Proceedings of the First International Conference on Embedded Networked Sensor Systems*, Los Angeles CA, Nov 2003.

[2] ACM. *Proc. of the Second International Conference on Embedded Networked Sensor Systems*, Nov 2004.

[3] M. Josie Ammer, Michael Sheets, Turan Karalar, Mika Kuulusa, and Jan Rabaey. A low-energy chip-set for wireless intercom. DAC, Jun 2003.

[4] Todd Austin, Eric Larson, and Dan Ernst. Simplescalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, 2002.

[5] Allan Beaufour, Martin Leopold, and Philippe Bonnet. Smart-tag based data dissemination. In *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications (WSNA02)*, pages 68–77, Jun 2002.

[6] Nicholas Beck and Ian Johnson. Shaping tinyos to deal with evolving device architectures: experiences porting tinyos-2.0 to the chipcon cc2430. In *EmNets '07: Proceedings of the 4th workshop on Embedded networked sensors*, pages 83–87, New York, NY, USA, 2007. ACM Press.

[7] J. Beutel, M. Dyer, M. Hinz, L. Meier, and M. Ringwald. Next-generation prototyping of sensor networks. pages 291–292. ACM Press, New York, Nov 2004.

[8] J. Beutel, O. Kasten, and M. Ringwald. Btnodes — a distributed platform for sensor nodes. pages 292–293. ACM Press, New York, Nov 2003.

[9] Jan Beutel. Metrics for sensor network platforms. In *Proc. ACM Workshop on Real-World Wireless Sensor Networks (REALWSN 06)*, pages 26–30. ACM Press, New York, June 2006.

[10] Jan Beutel, Matthias Dyer, Mustafa Yuecel, and Lothar Thiele. Development and test with the deployment-support network. In *Proc. 4th European Conf. on Wireless Sensor Networks (EWSN 2007), adjunct poster/demo proceedings*, pages 47–48. Parallel and Distributed Systems, TU Delft, The Netherlands, PDS-2007-001, January 2007.

[11] Jan Beutel and Oliver Kasten. A minimal bluetooth-based computing and communication platform. Technical report, ETH Zürich, May 2001.

[12] A. Boulis and M. B. Srivastava. Design and implementation of a framework for efficient and programmable sensor networks. In *Proc. of the First International Conference on Mobile Systems, Applications, and Services (MobiSys 2003), San Francisco, CA, USA*, San Francisco, CA, USA, May 2003. USENIX.

[13] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *ISCA*, pages 83–94, 2000. Available from: `citeseer.ist.psu.edu/brooks00wattch.html`.

[14] Fred Burghardt, Susan Mellers, and Jan Rabaey. The picoradio test bed. White Paper, Dec 2002.

[15] A. Chandrakasan, S. Sheng, and R. Brodersen. Low-power cmos digital design, 1992. Available from: `citeseer.ist.psu.edu/chandrakasan95low.html`.

[16] Sukwon Choi, Hojung Cha, and SungChil Cho. A soc-based sensor node: Evaluation of retos-enabled cc2430. In *Sensor, Mesh and Ad Hoc Communications and Networks, 2007. SECON '07. 4th Annual IEEE Communications Society Conference on*, pages 132–141. Department of Computer Science, Yonsei University, Jun 2007. Available from: `http://retos.yonsei.ac.kr/publications/cc2430-final-cr.pdf`.

[17] Sinem Coleri, Mustafa Ergen, and T. John Koo. Lifetime analysis of a sensor network with hybrid automata modelling. In *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 98–104. ACM Press, 2002.

[18] International Electrotechnical Commision. Letter symbols to be used in electrical technology - part 2: Telecommunications and electronics. IEC standard 60027-2, 2.nd edition, 2000. Available from: `https://domino.iec.ch/webstore/webstore.nsf/artnum/026554`.

[19] Cécile Cornou. *Automated Monitoring Methods For Group Housed Sows*. PhD thesis, Dept. of Large Animal Sciences, University of Copenhagen, Apr 2007.

[20] J. L. da Silva Jr., J. Shamberger, M. J. Ammer, C. Guo, S. Li, R. Shah, Tuan, M. Sheets, J. M. Rabaey, B. Nokolic, A. Sangiovanni-Vincentelli, and P. Wright. Design methodology for picoradio networks. IEEE Computer Society, 2001.

[21] Ashutosh Dhodapkar, Chee How Lim, George Cai, and W. Robert Daasch. Tem2p2est: A thermal enabled multi-model power/performance estimator. In *Proceedings of the First International Workshop on Power-Aware Computer Systems-Revised Papers*, pages 112–125. Springer-Verlag, 2001.

[22] Robert P. Dick, Ganesh Lakshminarayana, Anand Raghunathan, and Niraj K. Jha. Power analysis of embedded operating systems. In *Proceedings of the 37th conference on Design automation*, pages 312–315. ACM Press, 2000.

[23] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (Emnets-I)*, Tampa, Florida, USA, November 2004. Available from: `http://www.sics.se/~adam/dunkels04contiki.pdf`.

[24] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the Fourth ACM Conference on Embedded Networked Sensor Systems (SenSys 2006)*, November 2006. Available from: `http://www.sics.se/~adam/dunkels06protothreads.pdf`.

[25] Virantha Ekanayake, IV Clinton Kelly, and Rajit Manohar. An ultra low-power processor for sensor networks. In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 27–36. ACM Press, 2004.

[26] Jason Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Symposium on Operating Systems Principles*, pages 48–63, 1999. Available from: `citeseer.ist.psu.edu/flinn99energyaware.html`.

[27] Jason Flinn and M. Satyanarayanan. Powerscope: a tool for profiling the energy usage of mobile applications. In *Second IEEE Workshop on Mobile Computing Systems and Applications*, pages 2–10, February 1999. Available from: `citeseer.ist.psu.edu/flinn99powerscope.html`.

[28] Inc. Freescale Semiconductor. Mc13192 evaluation board reference manual, rev. 0.0. Data Sheet, 2004.

[29] Inc. Freescale Semiconductor. Sensor applicaion reference design, rev. 1.3. Data Sheet, 2004.

[30] Jerry Frenkil. Tools and methodologies for low power design. In *Design Automation Conference*, pages 76–81, 1997. Available from: `citeseer.ist.psu.edu/frenkil97tools.html`.

[31] David Gay and Jonathan Hui. Tep103: Permanent data storage (flash). TinyOS Enhancement Proposal. Available from: `http://tinyos.cvs.sourceforge.net/*checkout*/tinyos/tinyos-2.x/doc/html/tep103.html`.

[32] David Gay, Philip Levis, and David Culler. Software design patterns for tinyos. *Trans. on Embedded Computing Sys.*, 6(4):22, 2007.

[33] Lewis Girod, Thanos Stathopoulos, Nithya Ramanathan, Jeremy Elson, Deborah Estrin, Eric Osterweil, and Tom Schoellhammer. A system for simulation, emulation, and deployment of heterogeneous sensor networks. In *Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems*, Baltimore, MD, 2004. ACM.

[34] Omprakash Gnawali, Ki-Young Jang, Jeongyeup Paek, Marcos Vieira, Ramesh Govindan, Ben Greenstein, August Joki, Deborah Estrin, and Eddie Kohler. The tenet architecture for tiered sensor networks. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 153–166, New York, NY, USA, 2006. ACM.

[35] David J. Griffiths. *Introduction to electrodynamics*. Prentice Hall, third edition, 1999.

[36] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization*, Dec 2001.

[37] Chih-Chieh Han, Ram Kumar, Roy Shea, Eddie Kohler, and Mani Srivastava. A dynamic operating system for sensor nodes. In *MobiSys '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services*, pages 163–176, New York, NY, USA, 2005. ACM.

[38] V. Handziski, J.Polastre, J.H.Hauer, C.Sharp, A.Wolisz, and D.Culler. Flexible hardware abstraction for wireless sensor networks. *Proceedings of the 2nd European Workshop on Wireless Sensor Networks (EWSN 2005)*, Jan 2005.

[39] John S. Heidemann, Fabio Silva, Chalermek Intanagonwiwat, Ramesh Govindan, Deborah Estrin, and Deepak Ganesan. Building efficient wireless sensor networks with low-level naming. In *Symposium on Operating Systems Principles*, pages 146–159, 2001. Available from: `citeseer.ist.psu.edu/heidemann01building.html`.

[40] Teresia Heiskanen, Anders Ringgaard Kristensen, and Cécile Cornou. Trådløs overvågning - fremtidens værktøj i løsdriftsstalden. *Hyologisk*, pages 31–33, Jun 2005.

[41] Mark Hempstead, Matt Welsh, and David Brooks. Tinybench: The case for a standardized benchmark suite for tinyos based wireless sensor network devices. In *LCN '04: Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, pages 585–586, Washington, DC, USA, 2004. IEEE Computer Society.

[42] John L. Hennessy and David A. Patterson. *Computer Architecture a Quantitative Approach*. Morgan Kaufmann, 2nd edition, 1996.

[43] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David E. Culler, and Kristofer S. J. Pister. System architecture directions for networked sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000. Available from: `citeseer.nj.nec.com/382595.html`.

[44] Jason L. Hill and David E. Culler. Mica: A wireless platform for deeply embedded networks. *IEEE Micro*, 22(6):12–24, 2002.

[45] Jason Lester Hill. *System Architecture for Wireless Sensor Networks*. PhD thesis, University of California, Berkeley, 2003.

[46] Seth Edward-Austin Hollar. Cots dust. Master's thesis, University of California, Berkeley, 2000.

[47] Clinton Kelly IV, Virantha Ekanayake, and Rajit Manohar. Snap: A sensor-network asynchronous processor. In *Proceedings of the 9th International Symposium on Asynchronous Circuits and Systems*, page 24. IEEE Computer Society, 2003.

[48] Xiaofan Jiang, Prabal Dutta, David Culler, and Ion Stoica. Micro power meter for energy monitoring of wireless sensor networks at scale. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, pages 186–195, New York, NY, USA, 2007. ACM.

[49] Philo Juang, Hidekazu Oki, Yong Wang, Margaret Martonosi, Li Shiuan Peh, and Daniel Rubenstein. Energy-efficient computing for wildlife tracking: design tradeoffs and early experiences with zebranet. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 96–107. ACM Press, 2002.

[50] Nicolai Ascanius Jørgensen. Design of a low-power platform for running an embedded operating system. Master's thesis, DTU, Nov 2003.

[51] J. M. Kahn, R. H. Katz, and K. S. J. Pister. Next century challenges: Mobile networking for "smart dust". In *International Conference on Mobile Computing and Networking (MOBICOM)*, pages 271–278, 1999. Available from: `citeseer.ist.psu.edu/kahn99next.html`.

[52] R. Kling, R. Adler, J. Huang, V. Hummel, and L. Nachman. Intel mote: Using bluetooth in sensor networks. page 318. ACM Press, New York, Nov 2004.

[53] Peter Voigt Knudsen and Jan Madsen. Integrating communication protocol selection with partitioning in hardware/software codesign. In *ISSS '98: Proceedings of the 11th international symposium on System synthesis*, pages 111–116. IEEE Computer Society, 1998.

[54] Hans-Jörg Körber, Housam Wattar, Gerd Scholl, and Wolfgang Heller. Poster abstract: Embedding a microchip pic18f452 based commercial platform into tinyos. In *Proceedings of REALWSN 2005, Stockholm, Sweden, 20. - 21. June 2005*, June 2005. Available from: `http://www.sics.se/realwsn05/papers/korber05embedding.pdf`.

[55] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *International Symposium on Microarchitecture*, pages 330–335, 1997. Available from: `citeseer.ist.psu.edu/lee97mediabench.html`.

[56] Edward A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[57] Martin Leopold. Power estimation using the hogthrob prototype platform. Master's thesis, Dept. of Computer Science, University of Copenhagen, December 2004. Available from: `"http://www.distlab.dk/public/distsys/publications.php?id=49"`.

[58] Martin Leopold. Creating a new platform for tinyos 2.x. Technical Report 07/09, Dept. of Computer Science, University of Copenhagen, Sep 2007. Available from: `http://www.diku.dk/publikationer/tekniske.rapporter/rapporter/07-09.pdf`.

[59] Martin Leopold. Hogthrobv0 users manual. Technical Report 07/05, Dept. of Computer Science, University of Copenhagen, Sep 2007. Available from: `http://www.diku.dk/publikationer/tekniske.rapporter/rapporter/07-05.pdf`.

[60] Martin Leopold, Marcus Chang, and Philippe Bonnet. Prototyping wireless sensor network applications with btnodes. In *Proc. of the 5th European Workshop on Wireless Sensor Networks (EWSN)*. Springer-Verlag, January 2005.

[61] Martin Leopold, Mads Dydensborg, and Philippe Bonnet. Bluetooth and sensor networks: A reality check. In *Proceedings of the First International Conference on Embedded Networked Sensor Systems*, pages 103–113, November 2003. Available from: `http://www.distlab.dk/public/distsys/publications.php?id=38`.

[62] Philip Levis. Tinyos programming. Web book, Jun 2006. Available from: `http://csl.stanford.edu/~pal/pubs/tinyos-programming.pdf`.

[63] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. Tossim: accurate and scalable simulation of entire tinyos applications. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 126–137. ACM Press, 2003. Available from: `www.cs.berkeley.edu/~pal/pubs/tossim-sensys03.pdf`.

[64] Philip Levis, Sam Madden, David Gay, Joe Polastre, Robert Szewczyk, Alec Woo, Eric Brewer, and David Culler. The emergence of networking abstractions and techniques in tinyos. In *Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2004)*, 2004. Available from: `db.csail.mit.edu/madden/html/tinyos-nsdi04.pdf`.

[65] Xun Liu and Marios.C. Papaefthymiou. Hype: Hybrid power estimation for ip-based programmable systems. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, page january, 2003.

[66] Andreas Vad Lorentzen. Experimental sensor network: Lessons from the hogthrob project. Master's thesis, Dept. of Computer Science, University of Copenhagen, Apr 2006.

[67] Ciaran Lynch and Fergus O'Reilly. Pic-based tinyos implementation. In *Wireless Sensor Networks, 2005. Proceeedings of the Second European Workshop on*, pages 378–385, Feb 2005. Available from: `http://www.aws.cit.ie/Personnel/Papers/Paper261.pdf`.

[68] Ciarán Lynch and Fergus O'Reilly. Pic-based sensor operating system. In *proc. of Irish SIgnals and Signals Conference - ISSC 2004*, volume 2004, pages 95–100. IEE, 2004. Available from: `http://link.aip.org/link/abstract/IEECPS/v2004/iCP506/p95/s1`.

[69] Ciarán Lynch and Fergus O'Reilly. Processor choice for wireless sensor networks. In *REALWSN'05: Workshop on Real-World Wireless Sensor Networks*, pages 1–5, June 2005. Available from: `http://www.aws.cit.ie/Personnel/Papers/Paper263.pdf`.

[70] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tag: a tiny aggregation service for ad-hoc sensor networks. *SIGOPS Oper. Syst. Rev.*, 36(SI):131–146, 2002.

[71] Klaus Skelbæk Madsen. Experimental sensor network: Lessons from hogthrob. Master's thesis, Dept. of Computer Science, University of Copenhagen, 2006.

[72] Alan Mainwaring, Joseph Polastre, Robert Szewczyk, David Culler, and John Anderson. Wireless sensor networks for habitat monitoring. In *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 88–97, Atlanta, GA, September 2002. ACM Press. Available from: `citeseer.ist.psu.edu/mainwaring02wireless.html`.

[73] William P. McCartney and Nigamanth Sridhar. Abstractions for safe concurrent programming in networked embedded systems. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 167–180, New York, NY, USA, 2006. ACM.

[74] Shashidhar Mysore, Banit Agrawal, Frederic T. Chong, and Timothy Sherwood. Exploring the processor and isa design for wireless sensor network applications. In *Proceedings of the International Conference on VLSI Design*, Jan 2008.

[75] René Müller, Gustavo Alonso, and Donald Kossmann. Swissqm: Next generation data processing in sensor networks. In *Conference on Innovative Data Systems Research (CIDR)*, pages 1–9, 2007. Available from: `http://www.cidrdb.org/cidr2007/papers/cidr07p01.pdf`.

[76] L. Nazhandali, M. Minuth, and T. Austin. Sensebench: toward an accurate evaluation of sensor network processors. *iiswc*, 0:197–203, 2005.

[77] S. Neema, A. Mitra, A. Banerjee, W. Najjar, D. Zeinalipour-Yazti, D. Gunopulos, and V. Kalogeraki. Nodes: A novel system design for embedded sensor systems. Mentioned in Notable SPOTS platforms, IPSN 2005. Available from: `http://www.cs.ucr.edu/~najjar/papers/2005/nodes-paper.pdf`.

[78] Chulsung Park, Jinfeng Liu, and Pai H. Chou. Eco: an ultra-compact low-power wireless sensor node for real-time motion monitoring. In *IPSN '05: Proceedings of the 4th international symposium on Information processing in sensor networks*, page 54, Piscataway, NJ, USA, 2005. IEEE Press.

[79] Heemin Park, Weiping Liao, King Ho Tam, Mani B. Srivastava, and Lei He. A unified network and node level simulation framework for wireless sensor networks. Technical Report 7, CENS, Sep 2003.

[80] S. Park, A. Savvides, and M. B. Srivastava. Sensorsim: a simulation framework for sensor networks. In *Proceedings of the 3rd ACM international workshop on Modeling, analysis and simulation of wireless and mobile systems*, pages 104–111, Boston, MA USA, 2000.

[81] David Patnode, J. Dunne, A. Malinowski, and David R. Schertz. Wisenet - tinyos based wireless network of sensors. In *Industrial Electronics Society, 2003. IECON '03, The 29th Annual Conference of the IEEE*, pages 2363–2368, Nov 2003.

[82] Anders Egeskov Petersen, Sidsel Jensen, and Martin Leopold. Tep121: Towards tinyos for 8051. TinyOS Enhancement Proposal, Dec 2005. Available from: `http://tinyos.cvs.sourceforge.net/*checkout*/tinyos/tinyos-2.x/doc/html/tep121.html`.

[83] Joseph Polastre, Robert Szewczyk, Cory Sharp, and David Culler. The mote revolution: Low power wireless sensor network devices. In *Proceedings of IEEE HotChips 16*, Aug 2004.

[84] Joseph Robert Polastre. Design and implementation of wireless sensor networks for habitat monitoring. Master's thesis, University of California at Berkeley, 2003.

[85] Jonathan Polley, Dionysys Blazakis, Jonathan McGee, Dan Rusk, John S. Baras, and Manish Karir. Atemu: A fine-grained sensor network simulator. In *Proceedings of SECON'04, The First IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks*, 2004.

[86] D. Ponomarev, G. Kucuk, and K. Ghose. Accupower: An accurate power estimation tool for superscalar microprocessors. In *Proceedings of the conference on Design, automation and test in Europe*, page 124. IEEE Computer Society, 2002.

[87] Qinru Qiu, Qing Wu, Massoud Pedram, and Chih-Shun Ding. Cycle-accurate macro-models for rt-level power analysis. In *Proceedings of the 1997 international symposium on Low power electronics and design*, pages 125–130. ACM Press, 1997.

[88] J. Rabaey. *Digital Integrated Circuits A Design Perspective*. Prentice Hall, 1996.

[89] Jan M. Rabaey, M. Josie Ammer, Julio L. da Silva Jr., Danny Patel, and Shad Roundy. Picoradio supports ad hoc ultra-low power wireless networking. *IEEE Computer*, 33(7):42–48, Jul 2000.

[90] S. Ravi, A. Raghunathan, and S. T. Chakradhar. Efficient rtl power estimation for large designs. In *Proceedings of IEEE International Conference on VLSI Design*, Jan 2003.

[91] John Regehr, Alastair Reid, and Kirk Webb. Eliminating stack overflow by abstract interpretation. *Trans. on Embedded Computing Sys.*, 4(4):751–778, 2005.

[92] Margo Seltzer, David Krinsky, Keith Smith, and Xiaolan Zhang. The case for application-specific benchmarking. In *HOTOS '99: Proceedings of the The Seventh Workshop on Hot Topics in Operating Systems*, page 102. IEEE Computer Society, 1999. Available from: `citeseer.ist.psu.edu/seltzer99case.html`.

[93] Dongkun Shin, Hojun Shim, Yongsoo Joo, Han-Saem Yun, Jihong Kim, and Naehyuck Chang. Energy-monitoring tool for low-power embedded programs. *IEEE Design & Test of Computers*, 19(4):7–17, 2002.

[94] Victor Shnayder, Mark Hempstead, Bor rong Chen, Geoff Werner Allen, and Matt Welsh. Simulating the power consumption of large-scale sensor network applications. 2004.

[95] Tajana Simunic, Luca Benini, and Giovanni De Micheli. Cycle-accurate simulation of energy consumption in embedded systems. In *Design Automation Conference*, pages 867–872, 1999. Available from: `citeseer.ist.psu.edu/article/simunic99cycleaccurate.html`.

[96] Amit Sinha and Anantha Chandrakasan. Jouletrack — a web based tool for software energy profiling. In *Design Automation Conference*, pages 220–225, 2001. Available from: `citeseer.ist.psu.edu/sinha01jouletrack.html`.

[97] Sameer Sundresh, Wooyoung Kim, and Gul Agha. Sens: A sensor, environment and network simulator. University of Ilonios, IEEE, Apr 2004. Available from: `citeseer.ist.psu.edu/715442.html`.

[98] Robert Szewczyk, Alan Mainwaring, Joseph Polastre, John Anderson, and David Culler. An analysis of a large scale habitat monitoring application. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 214–226. ACM Press, 2004.

[99] T. Tan, A. Raghunathan, and N. Jha. Emsim: An energy simulation framework for an embedded operating system. In *Proceedings of International Symposium on Circuit and Systems*, May 2002. Available from: `citeseer.ist.psu.edu/tan02emsim.html`.

[100] Ben L. Titzer, Daniel K. Lee, and Jens Palsberg. Avrora: scalable sensor network simulation with precise timing. In *IPSN '05: Proceedings of the 4th international symposium on Information processing in sensor networks*, page 67, Piscataway, NJ, USA, 2005. IEEE Press.

[101] Tijs van Dam and Koen Langendoen. An adaptive energy-efficient mac protocol for wireless sensor networks. In *Proceedings of the First International Conference on Embedded Networked Sensor Systems*, pages 171–181, Los Angeles CA, Nov 2003. ACM Press.

[102] Kashif Virk, Martin Leopold, Andreas Vad Lorentzen, Martin Hansen, Phillipe Bonnet, and Jan Madsen. Design of a development platform for hw/sw codesign of wireless integrated sensor nodes. In *Eighth Euromicro Symposium on Digital Systems Design DSD 5*, pages 254–260, August-September 2005 2005.

[103] Geoffrey Werner-Allen, Patrick Swieskowski, and Matt Welsh. Motelab: a wireless sensor network testbed. In *IPSN '05: Proceedings of the 4th international symposium on Information processing in sensor networks*, page 68, Piscataway, NJ, USA, 2005. IEEE Press.

[104] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. The design and use of simplepower: a cycle-accurate energy estimation tool. In *Proceedings of the 37th conference on Design automation*, pages 340–345. ACM Press, 2000. Available from: `citeseer.ist.psu.edu/ye00design.html`.

[105] Pei Zhang, Christopher M. Sadler, Stephen A. Lyon, and Margaret Martonosi. Hardware design experiences in zebranet. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 227–238. ACM Press, 2004.